TURING 图灵程序设计丛书

[PACKT] PUBLISHING

【巴西】Loiane Groner 著　孙晓博 邓钢 吴双 陈迪 袁源 译

学习JavaScript数据结构与算法

Learning JavaScript Data Structures and Algorithms


中国工信出版集团


人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

Loiane Groner 花旗银行软件开发经理，负责海外项目的开发和团队管理；原IBM公司系统分析师及团队负责人；巴西坎皮纳斯Java用户组（CampinasJUG）领导者、圣埃斯皮里图Java用户组（ESJUG）协调人；巴西各大型技术会议特邀发言人；Sencha和Java技术布道者，通过博客（http://loianegroner.com）为软件开发社区撰稿，发表关于IT职业发展和常用开发技术的文章和视频。另著有《精通Ext JS》等书。

孙晓博 毕业于浙江大学，先后在百度、阿里任职。工作方向为移动前端，对WPO、组件化和移动站自动化测试有浓厚兴趣。微博：@筱卜D。

邓钢 非专业80后程序员，爱好Web技术。曾在盛大创新院担任前端工程师，现就职于IBM中国开发中心。除了不断磨练前端技术，工作之余对算法、数据结构等计算机科学的知识也很感兴趣。

吴双 九零后，苏宁易购前端工程师，毕业于华中科技大学。热爱前端，热爱生活。新浪微博：@吴双Orange。

陈迪 香港科技大学机械系硕士，机缘巧合进入前端领域，热爱前端和Web开发，目前在从事移动招聘的创业公司上海大岂网络负责前端开发。

袁源 毕业于厦门大学。2011年起从事前端开发，目前就职于百度。



【巴西】Loiane Groner 著 孙晓博 邓钢 吴双 陈迪 袁源 译

学习JavaScript数据结构与算法

Learning JavaScript Data Structures and Algorithms

人民邮电出版社

北 京

图书在版编目（C I P）数据

学习JavaScript数据结构与算法 / （巴西）格罗纳（Groner,L.）著 ; 孙晓博等译. -- 北京 : 人民邮电出版社, 2015.10
（图灵程序设计丛书）
ISBN 978-7-115-40414-5

Ⅰ. ①学… Ⅱ. ①格… ②孙… Ⅲ. ①数据结构②JAVA语言—程序设计 Ⅳ. ①TP311.12②TP312

中国版本图书馆CIP数据核字(2015)第218878号

内 容 提 要

本书首先介绍了 JavaScript 语言的基础知识，接下来讨论了数组、栈、队列、链表、集合、字典、散列表、树、图等数据结构，之后探讨了各种排序和搜索算法，包括冒泡排序、选择排序、插入排序、归并排序、快速排序、顺序搜索、二分搜索，最后还介绍了动态规划和贪心算法等常用的高级算法及相关知识。

本书适用于前端 Web 开发人员，以及所有对 JavaScript 数据结构与算法感兴趣的读者。

◆ 著　[巴西] Loiane Groner
译　孙晓博　邓　钢　吴　双　陈　迪　袁　源
责任编辑　岳新欣
执行编辑　仇祝平　李舒扬
责任印制　杨林杰

◆ 人民邮电出版社出版发行　北京市丰台区成寿寺路11号
邮编　100164　电子邮件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
北京　印刷

◆ 开本：800×1000　1/16
印张：10.75
字数：254千字　2015年 10 月第 1 版
印数：1 – 3 500册　2015年 10 月北京第 1 次印刷

著作权合同登记号　图字：01-2015-3277号

定价：39.00元

读者服务热线：(010)51095186转600　印装质量热线：(010)81055316

反盗版热线：(010)81055315

广告经营许可证：京崇工商广字第 0021 号

版 权 声 明

前　　言

JavaScript是当下最流行的编程语言。由于浏览器的原生支持（无需安装任何插件），JavaScript也被称作“互联网语言”。JavaScript的应用非常广泛，不仅被用于前端开发，也被用到服务器（Node.js）和数据库（MongoDB）环境中。

对任何专业技术人员来说，理解数据结构都非常重要。作为软件开发者，我们要能够用编程语言和数据结构来解决问题。编程语言和数据结构是这些问题解决方案中不可或缺的一部分。如果选择了不恰当的数据结构，可能会影响所写程序的性能。因此，了解不同数据结构和它们的适用范围十分重要。

算法在计算机科学中扮演着非常重要的角色。解决一个问题有很多种方法，但有些方法会比其他方法更好。因此，了解一下最著名的算法也很重要。

快乐地编码吧！

本书结构

第1章“JavaScript简介”，讲述了JavaScript的基础知识，它们可以帮助你更好地学习数据结构和算法，同时还介绍了如何搭建开发环境来运行书中的代码示例。

第2章“数组”，介绍了如何使用数组这种最基础且最常用的数据结构。这一章演示了如何对数组声明、初始化、添加和删除其中的元素，还讲述了如何使用JavaScript语言本身支持的数组方法。

第3章“栈”，介绍了栈这种数据结构，示范了如何创建栈，以及怎样添加和删除元素，还讨论了如何用栈解决计算机科学中的一些问题。

第4章“队列”，详述了队列这种数据结构，演示了如何创建队列，如何添加和删除队列中的元素，还讨论了如何用队列解决计算机科学中常见的问题，以及栈和队列的主要区别。

第5章“链表”，讲解如何用对象和指针从头创建链表这种数据结构。这一章除了讨论如何声明、创建、添加和删除链表元素之外，还介绍了不同类型的链表，例如双向链表和循环链表。

第6章“集合”，介绍了集合这种数据结构，讨论了如何用集合存储非重复性的元素。此外，还详述了对集合的各种操作以及相应代码的实现。

第7章“字典和散列表”，深入讲解字典、散列表及它们之间的区别。这一章介绍了这两种数据结构是如何声明、创建和使用的，还探讨了如何解决散列冲突，以及如何创建更高效的散列函数。

第8章“树”，讲解了树这种数据结构和它的相关术语，重点讨论了二叉搜索树，以及如何在树中搜索、遍历、添加和删除节点。如果想更深入地学习树（包括相关的算法），这一章还给出了一些建议。

第9章“图”，介绍了图这种数据结构和它的适用范围。这一章讲述了图的常用术语和不同表现方式，探讨了如何使用深度优先算法和广度优先算法遍历图，以及图的适用范围。

第10章“排序和搜索算法”，探讨了常用的排序算法，如冒泡排序（包括改进版）、选择排序、插入排序、归并排序和快速排序。另外还介绍了搜索算法中的顺序搜索和二分搜索。

第11章“算法补充知识”，主要讨论其他一些常用的算法和著名的大*O*表示法。这一章讲解了什么是递归，介绍了一些高级算法，如动态规划和贪心算法，还介绍了大*O*表示法和相关概念。最后，讨论了如何进一步学习算法的相关知识。

附录详尽列出了书中所授算法的复杂度列表（使用大*O*表示法）。

准备工作

为学习本书，你可以设置三种不同的开发环境。你不需要设置所有这三种环境，可以选择其一，也可以逐一尝试。

方法一，你需要一个浏览器，请在下面列表中选择其一：

- Chrome（https://www.google.com/chrome/browser/）
- Firefox（https://www.mozilla.org/en-US/firefox/new/）

方法二，你需要：

- 安装方法一中的任意一个浏览器；
- 安装一个Web服务器。如果你电脑里没有安装过Web服务器，推荐安装XAMPP（https://www.apachefriends.org）。

方法三，如果想安装一个纯JavaScript的环境，你需要完成下面几步。

- 安装步骤一中的任意浏览器

- 安装Node.js（http://nodejs.org/）
- 安装好Node.js后，安装`http-server`开发包：

```
npm install http-server -g
```

第1章还会对此进行更详细的介绍。

读者对象

本书的目标读者包括计算机科学专业的学生、刚刚开启职业生涯的技术人员，以及想学习基于JavaScript语言的数据结构和算法的朋友。如果想学好书中的数据结构和算法，编程知识和逻辑思维是必需的。

本书为数据结构和算法初学者所写，也为熟悉数据结构和算法，但想在JavaScript语言中使用它们的人所写。

排版约定

在本书中，你会发现一些不同的文本样式，用以区别不同种类的信息。下面举例说明。

正文中的代码、用户输入这样表示："在`script`标签里，编写JavaScript代码。"

代码段的格式如下：

```
console.log("num: "+ num);
console.log("name: "+ name);
console.log("trueValue: "+ trueValue);
console.log("price: "+ price);
console.log("nullVar: "+ nullVar);
console.log("und: "+ und);
```

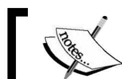
如果我们想让你重点关注代码段中的某个部分，会加粗显示：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>
    <script>
        alert('Hello, World!');
    </script>
</body>
</html>
```

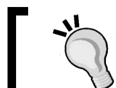
所有的命令行输入或输出的格式如下：

```
npm install http-server -g
```

新术语和重点词汇以楷体标示。屏幕、目录或对话框上的内容这样表示："**Node Packages Modules**（https://www.npmjs.org/）也在呈指数级增长。"



这个图标表示警告或需要特别注意的内容。



这个图标表示提示或者技巧。

读者反馈

欢迎提出反馈。如果你对本书有任何想法，喜欢它什么，不喜欢它什么，请让我们知道。要写出真正对大家有帮助的书，了解读者的反馈很重要。

一般的反馈，请发送电子邮件至feedback@packtpub.com，并在邮件主题中包含书名。

如果你有某个主题的专业知识，并且有兴趣写成或帮助促成一本书，请参考我们的作者指南http://www.packtpub.com/authors。

客户支持

现在，你是一位自豪的Packt图书的拥有者，我们会尽全力帮你充分利用你手中的书。

下载示例代码

你可以用你的账户从http://www.packtpub.com下载所有已购买Packt图书的示例代码文件。如果你从其他地方购买本书，可以访问http://www.packtpub.com/support并注册，我们将通过电子邮件把文件发送给你。

下载彩色插图

我们还提供了一份PDF文档，里面是本书中用到的彩色截图和图表。这些彩图能帮你更好地理解输出的变化。你可以从https://www.packtpub.com/sites/ default/files/downloads/4874OS_ColoredImages.pdf下载。

勘误表

虽然我们已尽力确保本书内容正确，但出错仍旧在所难免。如果你在我们的书中发现错误，不管是文本还是代码，希望能告知我们，我们不胜感激。这样做可以减少其他读者的困扰，帮助我们改进本书的后续版本。如果你发现任何错误，请访问http://www.packtpub.com/submit-errata提交，选择你的书，点击勘误表提交表单的链接，并输入详细说明。勘误一经核实，你的提交将被接受，此勘误将上传到本公司网站或添加到现有勘误表。从http://www.packtpub.com/support选择书名就可以查看现有的勘误表。

侵权行为

互联网上的盗版是所有媒体都要面对的问题。Packt非常重视保护版权和许可证。如果你发现我们的作品在互联网上被非法复制，不管以什么形式，都请立即为我们提供位置地址或网站名称，以便我们可以寻求补救。

请把可疑盗版材料的链接发到copyright@packtpub.com。

非常感谢你帮助我们保护作者，以及保护我们给你带来有价值内容的能力。

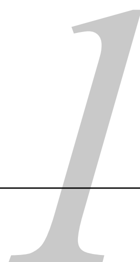
问题

如果你对本书内容存有疑问，不管是哪个方面，都可以通过questions@packtpub.com联系我们，我们将尽最大努力来解决。

目录

第 1 章

JavaScript简介



JavaScript是一门非常强大的编程语言。它是最流行的编程语言，也是网络应用里最卓越的语言之一。在GitHub（世界上最大的代码托管站点，https://github.com）上，托管了400 000多个JavaScript代码仓库（用JavaScript开发的项目数量也是最多的，参看http://goo.gl/ZFx6mg），并且还在逐年增长。

JavaScript不仅可用于前端开发，也适用于后端开发，Node.js就是这样一种技术。Node包（http://www.npmjs.org/）的数量也呈指数级增长。

要成为一名Web开发工程师，掌握JavaScript必不可少。

在本书中，你将学习最常用的数据结构和算法。为什么用JavaScript来学习这些数据结构和算法呢？我们已经回答了这个问题。JavaScript非常受欢迎，作为函数式编程语言，它非常适合用来学习数据结构和算法。通过它来学习数据结构比C或Java这些标准语言更简单，学习新东西也会变得很有趣。谁说数据结构和算法是只为C或Java这样的语言而生？在前端开发当中，你可能也需要实现它们。

学习数据结构和算法十分重要。首要原因是数据结构和算法可以很高效地解决常见问题，这对你今后的代码质量至关重要（也包括性能，要是用了不恰当的数据结构或算法，很可能会产生性能问题）。其次，对于计算机科学，算法是最基础的概念。最后，如果你想入职最好的IT公司（如谷歌、亚马逊、eBay等），数据结构和算法是面试问题的重头戏。

1.1 环境搭建

相比其他语言，JavaScript的优势之一在于不用安装或配置任何复杂的环境就可以开始学习。每台计算机上都已具备所需的环境，哪怕使用者从未写过一行代码。有浏览器足矣！

为了运行书中的示例代码，建议你做好如下准备：安装Chrome或Firefox浏览器（选择一个你最喜欢的即可），选择一个喜欢的编辑器（如Sublime Text），以及一个Web服务器（XAMPP或其他你喜欢的，这一步是可选的）。这些软件在Windows、Linux和Mac OS上均可以使用。

如果你使用Firefox，推荐你安装Firebug插件（https://getfirebug.com）。

接下来将介绍搭建环境的三种方案。

1.1.1　浏览器

浏览器是最简单的开发环境。

你也可以使用Firefox加Firebug。安装好Firebug后，在浏览器的右上角会看到如下图所示的图标。

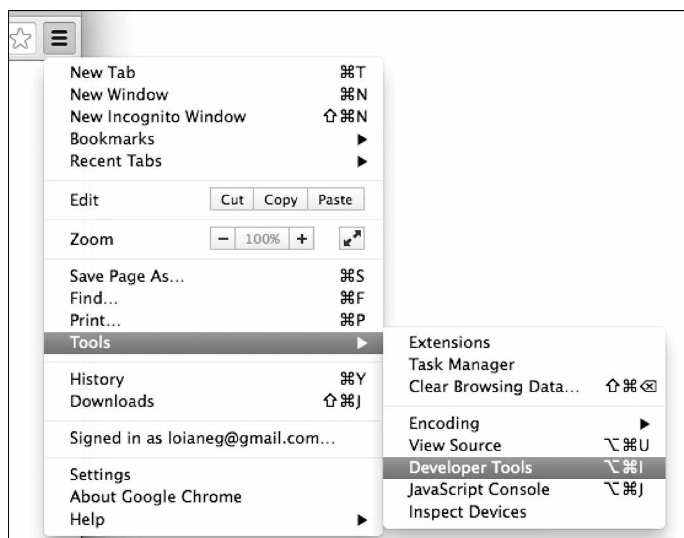


点击Firebug图标，打开它，可以看到Console标签，我们可以在其命令行区域中编写所有JavaScript代码，如下图所示（执行源代码请按Run按钮）。



也可以扩展命令行，来适应Firebug插件的整个可用区域。

你还可以使用谷歌Chrome，它已经集成了Google Developer Tools（谷歌开发者工具）。打开Chrome，点击设置及控制图标，选中Tools|Developer Tools，如下图所示。



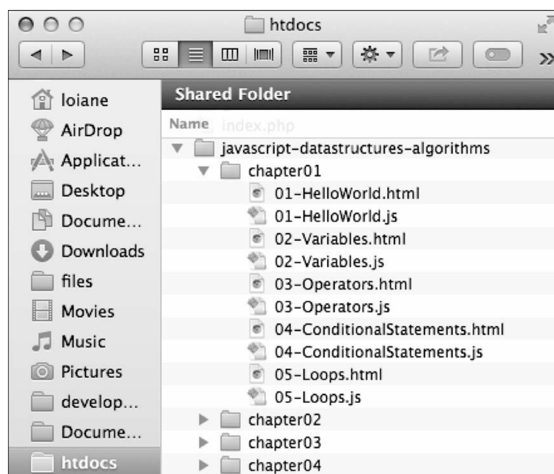
然后，就可以在Console标签页中编写JavaScript测试代码，如下所示。



1.1.2 使用Web服务器（XAMPP）

你可能想要安装的第二个环境是XAMPP，它的安装过程也很简单，但比只使用浏览器麻烦点儿。

安装XAMPP（https://www.apachefriends.org）或者你偏爱的其他Web服务器。然后，在XAMPP安装文件夹下找到htdocs目录。在该目录下新建一个文件夹，就可以在里面执行本书中所讲述的源代码；或者是直接将示例代码下载后提取到此目录，如下所示。



接下来，在启动XAMPP服务器之后，你就可以通过localhost这个URL，用浏览器访问源码，如下图所示（别忘了打开Firebug或谷歌开发者工具查看输出）。





执行示例代码时，请不要忘记打开谷歌开发者工具或Firebug查看输出结果。

1.1.3　使用Node.js搭建Web服务器

第三种选择就是100%的JavaScript，我们可以使用Node.js来搭建一个JavaScript服务器，不使用XAMPP搭建的Apache服务器。

首先要到http://nodejs.org/下载和安装Node.js。然后，打开终端应用（如果你用的是Windows操作系统，打开Node.js的命令行），输入如下命令：

```
npm install http-server -g
```

最好手动输入这些命令，复制粘贴可能会出错。

也可以用管理员身份执行上述命令。对于Linux和Mac操作系统，使用如下命令：

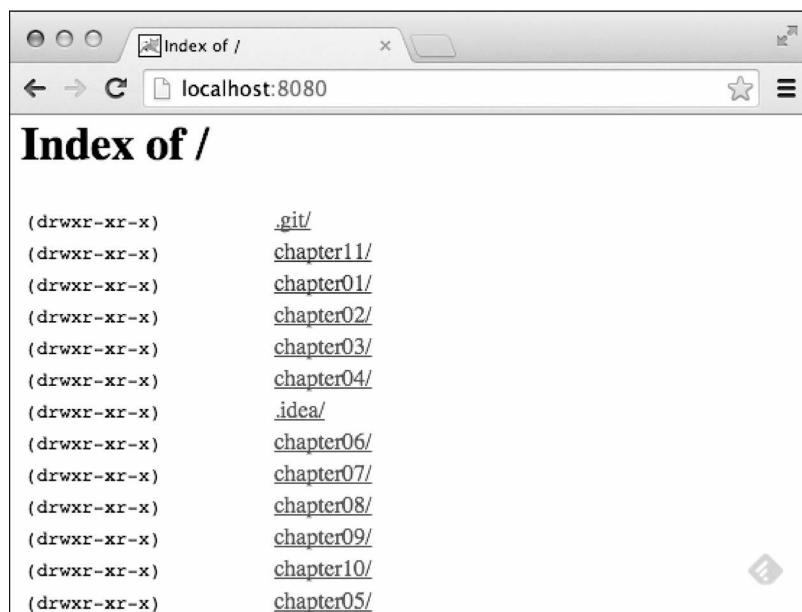
```
sudo npm install http-server -g
```

这条命令会在你的机器上安装一个JavaScript服务器：http-server。要启动服务器并在终端应用上运行本书中的示例代码，请将工作路径更改至示例代码文件夹，然后输入`http-server`，如下图所示，整个环境就搭建好了！

```
javascript-datastructures-algorithms — node — 82×7
loianeg:~ loiane$ cd /Users/loiane/Documents/javascript-datastructures-algorithms

loianeg:javascript-datastructures-algorithms loiane$ http-server
Starting up http-server, serving ./ on port: 8080
Hit CTRL-C to stop the server
```

为执行示例，打开浏览器，通过`http-server`命令指定的端口访问：





下载示例代码

在官网（http://www.packtpub.com）购买的所有Packt图书，均可以下载到对应的示例代码。如果你并非在官网购买的本书，请访问http://www. packtpub.com/support并注册你的邮箱，对应的代码文件就会发送给你。

本书的代码也可以在GitHub上找到，资源库地址为：https://github.com/loiane/javascriptdatastructures-algorithms。

1.2　JavaScript 基础

在深入学习各种数据结构和算法前，让我们先大概了解一下JavaScript。本节教大家一些相关的基础知识，有利于学习后面各章。

首先来看在HTML中编写JavaScript的两种方式：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>
    <script>
        alert('Hello, World!');
    </script>
</body>
</html>
```

第一种方式如上面的代码所示。创建一个HTML文件，把代码写进去。在这个例子里，我们在HTML中声明了`script`标签，然后把JavaScript代码都写进这个标签。

第二种方式，我们需要创建一个JavaScript文件（比如01-HelloWorld.js），在里面写入如下代码：

```
alert('Hello, World!');
```

然后，我们的HTML文件看起来如下：

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
</head>
<body>
    <script src="01-HelloWorld.js">
    </script>
</body>
</html>
```

第二个例子展示了如何将一个JavaScript文件引入HTML文件。

这两个例子，无论执行哪个输出都是一样的。但第二个例子是最佳实践。



可能你在网上的一些例子里看到过JavaScript的`include`语句，或者放在`head`标签中的JavaScript代码。作为最佳实践，我们会在关闭`body`标签前引入JavaScript代码。这样浏览器就会在加载脚本之前解析和显示HTML，有利于提升页面的性能。

1.2.1 变量

变量保存的数据可以在需要时设置、更新或提取。赋给变量的值都有对应的类型。JavaScript的类型有数字、字符串、布尔值、函数和对象。还有undefined和null，以及数组、日期和正则表达式。下面的例子介绍如何在JavaScript里使用变量。

```
var num = 1; //{1}
num = 3; //{2}

var price = 1.5; //{3}
var name = 'Packt'; //{4}
var trueValue = true; //{5}
var nullVar = null; //{6}
var und; //{7}
```

在行{1}，我们展示了如何声明一个JavaScript变量（声明了一个数字类型）。虽然关键字var不是必需的，但最好每次声明一个新变量时都加上。

在行{2}，我们更新了已有变量。JavaScript不是强类型语言。这意味着你可以声明一个变量并初始化成一个数字类型的值，然后把它更新成字符串或者其他类型的值，不过这并不是一个好做法。

在行{3}，我们又声明了一个数字类型的变量，不过这次是十进制浮点数。在行{4}，声明了一个字符串；在行{5}，声明了一个布尔值；在行{6}，声明了一个null；在行{7}，声明了undefined变量。null表示变量没有值，undefined表示变量已被声明，但尚未赋值：

```
console.log("num:" + num);
console.log("name:" + name);
console.log("trueValue:" + trueValue);
console.log("price:" + price);
console.log("nullVar:" +  nullVar);
console.log("und:" + und);
```

如果想看我们声明的每个变量的值，可以用console.log来实现，就像上面代码片段中那样。



> 书中示例代码会使用三种方式输出JavaScript的值。第一种是alert（'My text here'），将输出到浏览器的警示窗口；第二种是console.log（'My text here'），将把文本输出到调试工具的Console标签（谷歌开发者工具或是Firebug，根据你使用的浏览器而定）；第三种方式是直接输出到HTML页面里并被浏览器呈现，通过document.write（'My text here'）。可以选择你喜欢的方式来调试。

console.log方法能接收多个参数，除了console.log("num: " + num)还可以写成console.log("num: ", num)。

稍后我们会讨论函数和对象。

变量作用域

作用域指在编写的算法函数中，我们能访问的变量（在使用时，函数作用域也可以是一个函数）。有本地变量和全局变量两种。

让我们看一个例子：

```
var myVaribale = 'global';
myOtherVaribale = 'global';

function myFunction() {
    var myVariable = 'local';
    return myVaribale;
}

function myOtherFunction() {
    myOtherVariable = 'local';
    return myOtherVariable;
}

console.log(myVariable); //{1}
console.log(myFunction()); //{2}

console.log(myOtherVariable); //{3}
console.log(myOtherFunction()); //{4}
console.log(myOtherVariable); //{5}
```

行{1}输出global，因为它是一个全局变量。行{2}输出local，因为myVariable是在myFunction函数中声明的本地变量，所以作用域仅在myFunction内。

行{3}输出global，因为我们引用了在第二行初始化了的全局变量myOtherVariable。行{4}输出local。在myOtherFunction函数里，因为没有使用var关键字修饰，所以这里引用的是全局变量myOtherVariable并将它赋值为local。因此，行{5}会输出local（因为在myOtherFunction里修改了myOtherVariable的值）。

你可能听其他人提过在JavaScript里应该尽量少用全局变量，这是对的。通常，代码质量可以用全局变量和函数的数量来考量（数量越多越糟）。因此，尽可能避免使用全局变量。

1.2.2　操作符

编程语言里都需要操作符。在JavaScript里有算数操作符、赋值操作符、比较操作符、逻辑操作符、位操作符、一元操作符和其他操作符。我们来看一下这些操作符：

```
var num = 0; //{1}
num = num + 2;
```

```
num = num * 3;
num = num / 2;
num++;
num--;

num += 1; //{2}
num -= 2;
num *= 3;
num /= 2;
num %= 3;

console.log('num == 1 : ' + (num == 1)); // {3}
console.log('num === 1 : ' + (num === 1));
console.log('num != 1 : ' + (num != 1));
console.log('num > 1 : ' + (num > 1));
console.log('num < 1 : ' + (num < 1));
console.log('num >= 1 : ' + (num >= 1));
console.log('num <= 1 : ' + (num <= 1));

console.log('true && false : ' + (true && false)); // {4}
console.log('true || false : ' + (true || false));
console.log('!true : ' + (!true));
```

在行`{1}`，我们用了算数操作符。在下面的表格里，列出了这些操作符及其描述。

算数操作符	描　　述
+	加法
-	减法
*	乘法
/	除法
%	取余
++	递增
--	递减

在行`{2}`，我们使用了赋值操作符，在下面的表格里，列出了赋值操作符及其描述。

赋值操作符	描　　述
=	赋值
+=	加/赋值（x += y) == (x = x + y)
-=	减/赋值（x -= y) == (x = x - y)
*=	乘/赋值（x *= y) == (x = x * y)
/=	除/赋值（x /= y) == (x = x / y)
%=	取余/赋值（x %= y) == (x = x % y)

在行`{3}`，我们使用了比较操作符。在下面的表格里，列出了比较操作符及其描述。

比较操作符	描　　述
==	相等
===	全等
!=	不等
>	大于
>=	大于等于
<	小于
<=	小于等于

在行{4}，我们使用了逻辑操作符。在下面的表格里，列出了逻辑操作符及其描述。

逻辑操作符	描　　述
&&	与
||	或
!	非

JavaScript也支持位操作符，如下所示：

```
console.log('5 & 1:', (5 & 1));
console.log('5 | 1:', (5 | 1));
console.log('~ 5:', (~5));
console.log('5 ^ 1:', (5 ^ 1));
console.log('5 << 1:', (5 << 1));
console.log('5 >> 1:', (5 >> 1));
```

下面的表格对位操作符做了更详细的描述。

位操作符	描　　述
&	与
|	或
~	非
^	异或
<<	左移
>>	右移

typeof操作符可以返回变量或表达式的类型。我们看下面的代码：

```
console.log('typeof num:', typeof num);
console.log('typeof Packt:', typeof 'Packt');
console.log('typeof true:', typeof true);
console.log('typeof [1,2,3]:', typeof [1,2,3]);
console.log('typeof {name:John}:', typeof {name:'John'});
```

输出如下：

```
typeof num: number
typeof Packt: string
typeof true: boolean
typeof [1,2,3]: object
typeof {name:John}: object
```

JavaScript还支持delete操作符，可以删除对象里的属性：

```
var myObj = {name: 'John', age: 21};
delete myObj.age;
console.log(myObj); // 输出对象{name: "John"}
```

这些操作符在后面的算法学习中都会用到。

1.2.3 真值和假值

在JavaScript中，true和false有些复杂。在大多数编程语言中，布尔值true和false仅仅表示true/false。在JavaScript中，如"Packt"这样的字符串值，也可以看作true。

下面的表格能帮助我们更好地理解true和false在JavaScript中是如何转换的。

数值类型	转换成布尔值
undefined	false
null	false
布尔值	true是true，false是false
数字	+0、−0和NaN都是false，其他都是true
字符串	如果字符串是空的（长度是0）就是false，其他都是true
对象	true

我们来看一些代码，用输出来验证上面的总结：

```
function testTruthy(val){
    return val ? console.log('truthy') : console.log('falsy');
}

testTruthy(true); //true
testTruthy(false); //false
testTruthy(new Boolean(false)); //true (对象始终为true)

testTruthy(''); //false
testTruthy('Packt'); //true
testTruthy(new String('')); //true (对象始终为true)

testTruthy(1); //true
testTruthy(-1); //true
testTruthy(NaN); //false
testTruthy(new Number(NaN)); //true (对象始终为true)

testTruthy({}); //true (对象始终为true)
```

```
var obj = {name:'John'};
testTruthy(obj); //true
testTruthy(obj.name); //true
testTruthy(obj.age); //false (年龄不存在)
```

1.2.4　相等操作符（==和===）

当使用这两个相等操作符时，可能会引起一些困惑。

使用==时，不同类型的值也可以被看作相等。这样的结果可能会使那些资深的JavaScript开发者都感到困惑。我们用下面的表格给大家分析一下不同类型的值用相等操作符比较后的结果。

类型（x）	类型（y）	结　　果
null	undefined	true
undefined	null	true
数字	字符串	x == toNumber(y)
字符串	数字	toNumber(x) == y
布尔值	任何类型	toNumber(x) == y
任何类型	布尔值	x == toNumber(y)
字符串或数字	对象	x == toPrimitive(y)
对象	字符串或数字	toPrimitive(x) == y

如果x和y是相同类型，JavaScript会比较它们的值或对象值。其他没有列在这个表格中的情况都会返回false。

toNumber和toPrimitive方法是内部的，并根据以下表格对其进行估值。

toNumber方法对不同类型返回的结果如下：

值类型	结　　果
undefined	NaN
null	0
布尔值	如果是true，返回1；如果是false，返回0
数字	数字对应的值
字符串	将字符串解析成数字。如果字符串中包含字母，返回NaN；如果是由数字字符组成的，转换成数字
对象	Number(toPrimitive(vale))

toPrimitive方法对不同类型返回的结果如下：

值类型	结　　果
对象	如果对象的valueOf方法的结果是原始值，返回原始值；如果对象的toString方法返回原始值，就返回这个值；其他情况都返回一个错误

用例子来验证一下表格中的结果。首先，我们知道下面的代码输出true（字符串长度大于1）：

```
console.log('packt' ? true : false);
```

那么这行代码的结果呢?

```
console.log('packt' == true);
```

输出是false,为什么会这样呢?

(1) 首先,布尔值会被toNumber方法转成数字,因此得到packt == 1。

(2) 其次,用toNumber转换字符串值。因为字符串包含有字母,所以会被转成NaN,表达式就变成了NaN == 1,结果就是false。

那么这行代码的结果呢?

```
console.log('packt' == false);
```

输出也是false。步骤如下所示。

(1) 首先,布尔值会被toNumber方法转成数字,因此得到packt == 0。

(2) 其次,用toNumber转换字符串值。因为字符串包含有字母,所以会被转成NaN,表达式就变成了NaN == 0,结果就是false。

那么===操作符呢?简单多了。如果比较的两个值类型不同,比较的结果就是false。如果比较的两个值类型相同,结果会根据下表判断。

类型(*x*)	值	结 果
数字	*x*和*y*数值相同(但不是NaN)	true
字符串	*x*和*y*是相同的字符	true
布尔值	*x*和*y*都是true或false	true
对象	*x*和*y*引用同一个对象	true

如果*x*和*y*类型不同,结果就是false。

我们来看一些例子:

```
console.log('packt' === true); //false

console.log('packt' === 'packt'); //true

var person1 = {name:'John'};
var person2 = {name:'John'};
console.log(person1 === person2); //false,不同的对象
```

1.3 控制结构

JavaScript的控制结构和C与Java里的类似。条件语句支持if...else和switch。循环支持while、do...while和for。

1.3.1　条件语句

首先我们看一下如何构造if...else条件语句。有几种方式。

如果想让一个脚本仅当条件是true时执行，可以这样写：

```
var num = 1;
if (num === 1) {
    console.log("num is equal to 1");
}
```

如果想在条件为true的时候执行脚本A，其他情况下都执行脚本B，可以这样写：

```
var num = 0;
if (num === 1) {
    console.log("num is equal to 1");
} else {
    console.log("num is not equal to 1, the value of num is " +num);
}
```

if...else语句也可以用三元操作符替换，例如下面的if...else语句：

```
if (num === 1){
    num--;
} else {
    num++;
}
```

可以用三元操作符替换为：

```
(num === 1) ? num-- : num++;
```

如果我们有多个脚本，可以多次使用if...else，根据不同的条件执行不同的语句：

```
var month = 5;
if (month === 1) {
    console.log("January");
} else if (month === 2){
    console.log("February");
} else if (month === 3){
    console.log("March");
} else {
    console.log("Month is not January, February or March");
}
```

最后，还有switch语句。如果要判断的条件和上面的一样（但要和不同的值进行比较），可以使用swtich语句：

```
var month = 5;
switch(month) {
    case 1:
        console.log("January");
        break;
```

```
        case 2:
            console.log("February");
            break;
        case 3:
            console.log("March");
            break;
        default:
            console.log("Month is not January, February or March");
    }
```

对于switch语句来说，case和break关键字的用法很重要。case判断当前switch的值是否和case分支语句的值相等。break会中止switch语句的执行。没有break会导致执行完当前的case后，继续执行下一个case，直到遇到break或switch执行结束。最后，还有default关键字，在表达式不匹配前面任何一种情形的时候，就执行default中的代码（如果有对应的，就不会执行）。

1.3.2 循环

在处理数组元素时会经常用到循环（数组是下一章的主讲内容）。在我们的算法中也会经常用到for循环。

JavaScript中的for循环与C和Java中的一样。循环的计数值通常是一个数字，然后和另一个值比较（如果条件成立就会执行for循环中的代码），之后这个数值会递增或递减。

在下面的代码里，我们用了一个for循环。当i小于10时，会在控制台中输出其值。i的初始值是0，因此这段代码会输出0到9。

```
for (var i=0; i<10; i++) {
    console.log(i);
}
```

我们要关注的下一种循环是while循环。当while的条件判断成立时，会执行循环内的代码。下面的代码里，有一个初始值为0的变量i，我们希望在i小于10时输出它的值。输出会是0到9：

```
var i = 0;
while(i<10)
{
    console.log(i);
    i++;
}
```

do...while循环和while循环很相似。区别是在while循环里，先进行条件判断再执行循环体中的代码，而在do...while循环里，是先执行循环体中的代码再判断循环条件。do...while循环至少会让循环体中的代码执行一次。下面的代码同样会输出0到9：

```
var i = 0;
do {
```

```
    console.log(i);
    i++;
} while (i<10)
```

1.4　函数

在用JavaScript编程时，函数很重要。在我们的例子里也用了函数。

下面的代码展示了函数的基本语法。它没有用到参数或`return`语句：

```
function sayHello() {
    console.log('Hello!');
}
```

要执行这个函数，只需要这样调用一下：

```
sayHello();
```

我们也可以传递参数给函数。参数是会被函数使用的变量。下面的代码展示了如何在函数中使用参数：

```
function output(text) {
    console.log(text);
}
```

我们可以通过以下代码使用该函数：

```
output('Hello!');
```

你可以传递任意数量的参数，如下所示：

```
output('Hello!', 'Other text');
```

在这个例子中，函数只使用了传入的第一个参数，第二个参数被忽略。

函数也可以返回一个值，例如：

```
function sum(num1, num2) {
    return num1 + num2;
}
```

这个函数计算了给定两个数字之和，并返回结果。我们可以这样使用：

```
var result = sum(1,2);
output(result);
```

1.5　面向对象编程

JavaScript里的对象就是普通名值对的集合。创建一个普通对象有两种方式。第一种方式是：

```
var obj = new Object();
```

第二种方式是：

```
var obj = {};
```

也可以这样创建一个完整的对象：

```
obj = {
    name: {
        first: 'Gandalf',
        last: 'the Grey'
    },
    address: 'Middle Earth'
};
```

在面向对象编程（OOP）中，对象是类的实例。一个类定义了对象的特征。我们会创建很多类来表示算法和数据结构。例如我们声明了一个类来表示书：

```
function Book(title, pages, isbn){
    this.title = title;
    this.pages = pages;
    this.isbn = isbn;
}
```

用下面的代码实例化这个类：

```
var book = new Book('title', 'pag', 'isbn');
```

然后，我们可以访问和修改对象的属性：

```
console.log(book.title); //输出书名
book.title = 'new title'; //修改书名
console.log(book.title); //输出新的书名
```

类可以包含函数。可以声明和使用函数，如下所示：

```
Book.prototype.printTitle = function(){
    console.log(this.title);
};
book.printTitle();
```

也可以直接在类的定义里声明函数：

```
function Book(title, pages, isbn){
    this.title = title;
    this.pages = pages;
    this.isbn = isbn;
    this.printIsbn = function(){
        console.log(this.isbn);
    }
}
book.printIsbn();
```



在原型的例子里，printTitle方法只会创建一次，在Book类的所有实例中共享。如果是在定义类的内部结构时声明，每个类的实例都会有一份该方法的副本。使用原型方法可以节约内存和降低实例化的开销。最好在声明公共方法时使用基于原型的方法。生成私有方法时用在类定义时内部声明的方式，这样其他实例不会访问到这个方法。你可能注意到了，在上面的例子中我们是在定义类内部结构时声明方法（因为我想让这些属性和方法为各个实例单独拥有），但还是尽量使用基于原型的方法定义更好些。

现在，我们已经学完了本书所需的所有JavaScript基础知识，接下来就可以学习数据结构和算法了。

1.6　调试工具

除了学会如何用JavaScript编程外，还需要了解如何调试代码。调试对于找到代码中的错误十分有帮助，也能让你低速执行代码，看到所有发生的事情（方法被调用的栈、变量赋值等）。极力推荐你花一些时间学习一下如何调试书中的源码，查看算法的每一步（这样也会让你对算法有深刻的理解）。

Firefox和Chrome都支持调试。这里有一个了解谷歌开发者工具的好教程，地址是https://developer.chrome.com/devtools/docs/javascript-debugging。

除了你喜好的编辑器外，这里推荐其他几个工具，可以提升编写JavaScript的效率。

- **Aptana**：这是一个开源的免费IDE，支持JavaScript、CSS3和HTML5以及其他语言（http://www.aptana.com）。
- **WebStorm**：这是一个很强大的IDE，支持最新的Web技术和框架。它不是免费的，但你可以下载一个30天试用版本体验一下（http://www.jetbrain.com/webstorm）。
- **Sublime Text**：这是一个轻量级的文本编辑器，可以自定义插件。可以买它的许可证来支持这个工具的开发，也可以免费使用（试用版不过期），http://www.sublimetext.com/。

1.7　小结

本章主要讲述了如何搭建开发环境，有了这个环境就可以编写和运行书中的示例代码。

本章也讲了JavaScript语言的基础知识，这些知识会在接下来的数据结构和算法学习过程中用到。

下一章，我们要学习第一种数据结构：数组。许多语言都对数组有原生的支持（当然也包括JavaScript）。

第 2 章　数　组

几乎所有的编程语言都原生支持数组类型，因为数组是最简单的内存数据结构。JavaScript里也有数组类型，虽然它的第一个版本并没有支持数组。本章中，我们将深入学习数组数据结构和它的能力。

数组存储一系列同一种数据类型的值。但在JavaScript里，也可以在数组中保存不同类型的值。但我们还是要遵守最佳实践，别这么做（大多数语言都没这个能力）。

2.1　为什么用数组

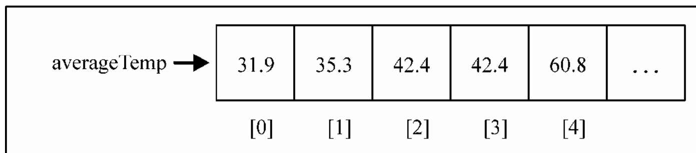
假如有这样一个需求：保存所在城市每个月的平均温度。可以这么做：

```
var averageTempJan = 31.9;
var averageTempFeb = 35.3;
var averageTempMar = 42.4;
var averageTempApr = 52;
var averageTempMay = 60.8;
```

当然，这肯定不是最好的方案。按照这种方式，如果只存一年的数据，我们能管理12个变量。但要多存几年的平均温度呢？幸运的是，我们可以用数组来解决，更加简洁地呈现同样的信息：

```
averageTemp[0] = 31.9;
averageTemp[1] = 35.3;
averageTemp[2] = 42.4;
averageTemp[3] = 52;
averageTemp[4] = 60.8;
```

数组`averageTemp`里的内容如下图所示：



2.2　创建和初始化数组

用JavaScript声明、创建和初始化数组很简单，就像下面这样：

```
var daysOfWeek = new Array(); //{1}
var daysOfWeek = new Array(7); //{2}
var daysOfWeek = new Array('Sunday', 'Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday', 'Saturday'); //{3}
```

使用new关键字，就能简单地声明并初始化一个数组（行{1}）。用这种方式，还可以创建一个指定长度的数组（行{2}）。另外，也可以直接将数组元素作为参数传递给它的构造器（行{3}）。

其实，用new创建数组并不是最好的方式。如果你想在JavaScript中创建一个数组，只用中括号（[]）的形式就行了，如下所示：

```
var daysOfWeek = [];
```

也可使用一些元素初始化数组，如下：

```
var daysOfWeek = ['Sunday', 'Monday', 'Tuesday', 'Wednesday',
'Thursday', 'Friday', 'Saturday'];
```

如果想知道数组里已经存了多少个元素，可以使用数组的length属性。以下代码的输出是7：

```
console.log(daysOfWeek.length);
```

要访问数组里特定位置的元素，可以用中括号传递数值位置，得到想知道的值或者赋新的值。假如我们想输出数组daysOfWeek里的所有元素，可以通过循环遍历数组，打印元素，如下所示：

```
for (var i=0; i<daysOfWeek.length; i++){
    console.log(daysOfWeek[i]);
}
```

我们来看另一个例子：求斐波那契数列的前20个数字。已知斐波那契数列中第一个数字是1，第二个是2，从第三项开始，每一项都等于前两项之和：

```
var fibonacci = []; //{1}
fibonacci[1] = 1; //{2}
fibonacci[2] = 1; //{3}

for(var i = 3; i < 20; i++){
    fibonacci[i] = fibonacci[i-1] + fibonacci[i-2]; ////{4}
}

for(var i = 1; i<fibonacci.length; i++){ //{5}
    console.log(fibonacci[i]);           //{6}
}
```

在行{1}处，我们声明并创建了一个数组。在行{2}和行{3}，把斐波那契数列中的前两个数字分别赋给了数组的第二和第三位（在JavaScript中，数组的第一位是0，这里我们略过，从第二

位开始分别保存斐波那契数列中对应位置的元素)。

然后，我们需要做的就是想办法得到斐波那契数列的第三到第二十位的数字(前两个值我们已经初始化过了)。我们可以用循环来处理，把数组中前两位上的元素相加，结果赋给当前位置上的元素(行{4}——从数组中的索引3到索引19)。

最后，看看输出(行{6})，我们只需要循环遍历数组的各个元素(行{5})。



示例代码里，我们用console.log来输出数组中对应索引位置的值(行{5}和行{6})，也可以直接用console.log(fibonacci)输出数组。大多数浏览器都可以用这种方式，清晰地输出数组。

现在如果想知道斐波那契数列其他位置上的值是多少，要怎么办呢？很简单，把之前循环条件中的终止变量从20改成你希望的值就可以了。

2.3 添加和删除元素

从数组中添加和删除元素也很容易，但有时也会很棘手。假如我们有一个数组numbers，初始化成0到9：

```
var numbers = [0,1,2,3,4,5,6,7,8,9];
```

如果想要给数组添加一个元素(比如10)，只要把值赋给数组中最后一个空位上的元素即可。

```
numbers[numbers.length] = 10;
```



在JavaScript中，数组是一个可以修改的对象。如果添加元素，它就会动态增长。在C和Java等其他语言里，我们要决定数组的大小，想添加元素就要创建一个全新的数组，不能简单地往其中添加所需的元素。

另外，还有一个push方法，能把元素添加到数组的末尾。通过push方法，能添加任意个元素：

```
numbers.push(11);
numbers.push(12, 13);
```

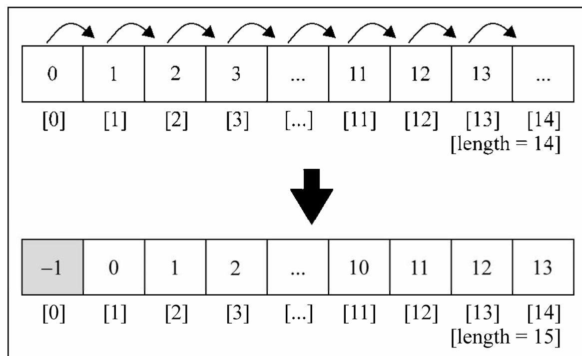
如果输出numbers的话，就会看到从0到13的值。

现在，我们希望在数组中插入一个值，不像之前那样插入到最后，而是放到数组的首位。为了实现这个需求，首先我们要腾出数组里第一个元素的位置，把所有的元素向右移动一位。我们可以循环数组中的元素，从最后一位+1 (长度)开始，将其对应的前一个元素的值赋给它，依

次处理，最后把我们想要的值赋给第一个位置（–1）上。

```
for (var i=numbers.length; i>=0; i--){
    numbers[i] = numbers[i-1];
}
numbers[0] = -1;
```

下面这张图描述了我们刚才的操作过程：



在JavaScript里，数组有一个方法叫`unshift`，可以直接把数值插入数组的首位：

```
numbers.unshift(-2);
numbers.unshift(-4, -3);
```

那么，用`unshift`方法，我们就可以在数组的开始处添加值–2， 然后添加–3、–4等。这样数组就会输出数字–4到13。

目前为止，我们已经学习了如何给数组的开始和结尾位置添加元素。下面我们来看一下怎样从数组中删除元素。

要删除数组里最靠后的元素，可以用`pop`方法：

```
numbers.pop();
```



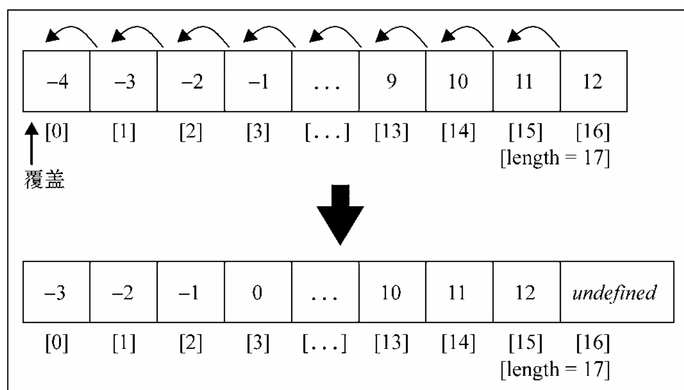
通过push和pop方法，就能用数组来模拟栈，你将会在下一章看到这部分内容。

现在，数组输出的数字是–4到12，并且数组的长度是17。

如果要移除数组里的第一个元素，可以用下面的代码：

```
for (var i = 0, l < numbers.length; i++){
    numbers[i] = numbers[i+1];
}
```

下面这张图呈现了这段代码的执行过程：



我们把数组里所有的元素都左移了一位。但数组的长度依然是17，这意味着数组中有额外的一个元素（值是undefined）。在最后一次循环里，i + 1引用了一个数组里还未初始化的位置（在其他一些语言里，这样写可能就会抛出异常了，因此不得不在numbers.length - 1处停止循环）。

可以看到，我们只是把数组第一位的值用第二位覆盖了，并没有删除元素（因为数组的长度和之前还是一样的，并且了多一个未定义元素）。

要确实删除数组的第一个元素，可以用shift方法实现：

```
numbers.shift();
```

那么，假如本来数组中的值是从–4到12，长度为17，执行了上述代码后，数组就只有–3到12了，并且长度也会减小到16。



通过shift和unshift方法，就能用数组模拟基本的队列数据结构，第4章里会讲到。

目前为止，我们已经学习了如何添加元素到数组的开头或结尾处，以及怎样删除数组开头和结束位置上的元素。那如何在数组中的任意位置上删除或添加元素？

我们可以使用splice方法，简单地通过指定位置/索引，就可以删除相应位置和数量的元素：

```
numbers.splice(5,3);
```

这行代码删除了从数组索引5开始的3个元素。这就意味着numbers[5]、numbers[6]和numbers[7]从数组中删除了。现在数组里的值变成了–3、–2、–1、0、1、5、6、7、8、9、10、11和12（2、3、4已经被移除）。

现在，我们想把数字2、3、4插入数组里，放到之前删除元素的位置上，可以再次使用splice方法：

```
numbers.splice(5,0,2,3,4);
```

splice方法接收的第一个参数，表示想要删除或插入的元素的索引值。第二个参数是删除元素的个数（这个例子里，我们的目的不是删除元素，所以传入0）。第三个参数往后，就是要添加到数组里的值（元素2、3、4）。输出会发现值又变成了从–3到12。

最后，执行下这行代码：

```
numbers.splice(5,3,2,3,4);
```

输出的值是从–3到12。原因在于，我们从索引5开始删除了3个元素，但也从索引5开始添加了元素2、3、4。

2.4　二维和多维数组

还记得本章开头平均气温测量的例子吗？现在我打算再用一下，不过把记录的数据改成数天内每小时的气温。现在我们已经知道可以用数组来保存这些数据，那么要保存两天的每小时气温数据就可以这样：

```
var averageTempDay1 = [72,75,79,79,81,81];
var averageTempDay2 = [81,79,75,75,73,72];
```

然而，这不是最好的方法。我们可以做得更好。我们可以使用矩阵（二维数组）来存储这些信息。矩阵的行保存每天的数据，列对应小时级别的数据：

```
var averageTemp = [];
averageTemp[0] = [72,75,79,79,81,81];
averageTemp[1] = [81,79,75,75,73,72];
```

JavaScript只支持一维数组，并不支持矩阵。但是，我们可以像上面的代码一样，用数组套数组，实现矩阵或任一多维数组。代码也可以写成这样：

```
//day 1
averageTemp[0] = [];
averageTemp[0][0] = 72;
averageTemp[0][1] = 75;
averageTemp[0][2] = 79;
averageTemp[0][3] = 79;
averageTemp[0][4] = 81;
averageTemp[0][5] = 81;
//day 2
averageTemp[1] = [];
averageTemp[1][0] = 81;
averageTemp[1][1] = 79;
```

```
averageTemp[1][2] = 75;
averageTemp[1][3] = 75;
averageTemp[1][4] = 73;
averageTemp[1][5] = 72;
```

上面的代码里，我们分别指定了每天和每小时的数据。数组中的内容如下图所示：

	[0]	[1]	[2]	[3]	[4]	[5]
[0]	72	75	79	79	81	81
[1]	81	79	75	75	73	73

每行就是每天的数据，每列是当天不同时段的气温。

如果想看这个矩阵的输出，我们可以创建一个通用函数，专门输出其中的值：

```
function printMatrix(myMatrix) {
    for (var i=0; i<myMatrix.length; i++){
        for (var j=0; j<myMatrix[i].length; j++){
            console.log(myMatrix[i][j]);
        }
    }
}
```

需要遍历所有的行和列。因此，我们需要使用一个嵌套的`for`循环来处理，其中变量`i`为行，变量`j`为列。

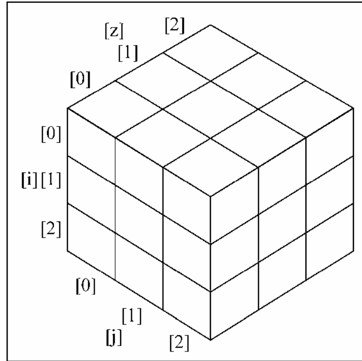
使用以下代码查看矩阵`averageTemp`的输出：

```
printMatrix(averageTemp);
```

以此类推，也可以用这种方式来处理多维数组。假如我们要创建一个3×3的矩阵，每一格里包含矩阵的`i`（行）、`j`（列）及`z`（深度）之和：

```
var matrix3x3x3 = [];
for (var i=0; i<3; i++){
    matrix3x3x3[i] = [];
    for (var j=0; j<3; j++){
        matrix3x3x3[i][j] = [];
        for (var z=0; z<3; z++){
            matrix3x3x3[i][j][z] = i+j+z;
        }
    }
}
```

数据结构中有几个维度都没关系，我们都可以用循环遍历每个维度来访问所有格子。3×3×3的矩阵也可用立体图表示如下：



可以用以下代码输出这个矩阵的内容：

```
for (var i=0; i<matrix3x3x3.length; i++){
    for (var j=0; j<matrix3x3x3[i].length; j++){
        for (var z=0; z<matrix3x3x3[i][j].length; z++){
            console.log(matrix3x3x3[i][j][z]);
        }
    }
}
```

如果是一个3×3×3×3的矩阵，代码中就会用四层嵌套的for语句，以此类推。

2.5　JavaScript 的数组方法参考

在JavaScript里，数组是可修改的对象，这意味着创建的每个数组都有一些可用的方法。数组很有趣，因为它们十分强大，并且相比其他语言中的数组，JavaScript中的数组有许多很好用的方法。这样就不用再为它开发一些基本功能了，例如在数据结构的中间添加或删除元素。

下面的表格中详述了数组的一些核心方法，其中的一些我们已经学习过了。

方法名	描　　述
concat	连接2个或更多数组，并返回结果
every	对数组中的每一项运行给定函数，如果该函数对每一项都返回true，则返回true
filter	对数组中的每一项运行给定函数，返回该函数会返回true的项组成的数组
forEach	对数组中的每一项运行给定函数。这个方法没有返回值
join	将所有的数组元素连接成一个字符串
indexOf	返回第一个与给定参数相等的数组元素的索引，没有找到则返回-1
lastIndexOf	返回在数组中搜索到的与给定参数相等的元素的索引里最大的值
map	对数组中的每一项运行给定函数，返回每次函数调用的结果组成的数组
reverse	颠倒数组中元素的顺序，原先第一个元素现在变成最后一个，同样原先的最后一个元素变成了现在的第一个
slice	传入索引值，将数组里对应索引范围内的元素作为新数组返回

（续）

方法名	描　述
some	对数组中的每一项运行给定函数，如果任一项返回true，则返回true
sort	按照字母顺序对数组排序，支持传入指定排序方法的函数作为参数
toString	将数组作为字符串返回
valueOf	和toString类似，将数组作为字符串返回

我们已经学过了push、pop、shift、unshift和splice方法。下面来看表格中提到的方法。在本书接下来的章节里，编写数据结构和算法时会大量用到这些方法。

2.5.1　数组合并

考虑如下场景：有多个数组，需要合并起来成为一个数组。我们可以迭代各个数组，然后把每个元素加入最终的数组。幸运的是，JavaScript已经给我们提供了解决方法，叫作concat方法：

```
var zero = 0;
var positiveNumbers = [1,2,3];
var negativeNumbers = [-3,-2,-1];
var numbers = negativeNumbers.concat(zero, positiveNumbers);
```

concat方法可以向一个数组传递数组、对象或是元素。数组会按照该方法传入的参数顺序连接指定数组。在这个例子里，zero将被合并到nagativeNumbers中，然后positiveNumbers继续被合并。最后输出的结果是–3、–2、–1、0、1、2、3。

2.5.2　迭代器函数

有时我们需要迭代数组中的元素。前面我们已经学过，可以用循环语句来处理，例如for语句。

JavaScript内置了许多数组可用的迭代方法。对于本节的例子，我们需要数组和函数。假如有一个数组，它值是从1到15，如果数组里的元素可以被2整除（偶数），函数就返回true，否则返回false：

```
var isEven = function (x) {
    // 如果x是2的倍数，就返回true
    console.log(x);
    return (x % 2 == 0) ? true : false;
    // 也可以写成return (x % 2 == 0) ? true : false
};
var numbers = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15];
```

return (x % 2 == 0) ? true : false 也可以写成return (x % 2== 0)。

我们要尝试的第一个方法是every。every方法会迭代数组中的每个元素，直到返回false。

```
numbers.every(isEven);
```

在这个例子里，数组numbers的第一个元素是1，它不是2的倍数（1是奇数），因此isEven 函数返回false，然后every执行结束。

下一步，我们来看some方法。它和every的行为类似，不过some方法会迭代数组的每个元素，直到函数返回true：

```
numbers.some(isEven);
```

在我们的例子里，numbers数组中第一个偶数是2（第二个元素）。第一个被迭代的元素是1，isEven会返回false。第二个被迭代的元素是2，isEven返回true——迭代结束。

如果要迭代整个数组，可以用forEach方法。它和使用for循环的结果相同：

```
numbers.forEach(function(x){
    console.log((x % 2 == 0));
});
```

JavaScript还有两个会返回新数组的遍历方法。第一个是map：

```
var myMap = numbers.map(isEven);
```

数组myMap里的值是：[false, true, false, true, false, true, false, true, false, true, false, true, false, true, false]。它保存了传入map方法的isEven函数的运行结果。这样就很容易知道一个元素是否是偶数。比如，myMap[0]是false，因为1不是偶数；而myMap[1]是true，因为2是偶数。

还有一个filter方法。它返回的新数组由使函数返回true的元素组成：

```
var evenNumbers = numbers.filter(isEven);
```

在我们的例子里，evenNumbers数组中的元素都是偶数：[2, 4, 6, 8, 10, 12, 14]。

最后是reduce方法。reduce方法接收一个函数作为参数，这个函数有四个参数：previousValue、currentValue、index和array。这个函数会返回一个将被叠加到累加器的值，reduce方法停止执行后会返回这个累加器。如果要对一个数组中的所有元素求和，这就很有用，比如：

```
numbers.reduce(function(previous, current, index){
    return previous + current;
});
```

输出将会是120。

2.5.3　搜索和排序

通过本书，我们能学到如何编写最常用的搜索和排序算法。其实，JavaScript里也提供了一个

排序方法和一组搜索方法。让我们来看看。

首先，我们想反序输出数组numbers（它本来的排序是1, 2, 3, 4,⋯15）。要实现这样的功能，可以用reverse方法，然后数组内元素就会反序。

```
numbers.reverse();
```

现在，输出numbers的话就会看到[15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]。然后，我们用sort方法：

```
numbers.sort();
```

然而，如果输出数组，结果会是[1, 10, 11, 12, 13, 14, 15, 2, 3, 4, 5, 6, 7, 8, 9]。看起来不大对，是吧？这是因为sort方法在对数组做排序时，把元素默认成字符串进行相互比较。

我们可以传入自己写的比较函数，因为数组里都是数字，所以可以这样写：

```
numbers.sort(function(a, b){
    return a-b;
});
```

这段代码，对于b大于a时，会返回负数，反之则返回正数。如果相等的话，就会返回0。也就是说返回的是负数，就说明a比b小，这样sort就根据返回值的情况给数组做排序。

之前的代码也可以被表示成这样，会更清晰一些：

```
function compare(a, b) {
    if (a < b) {
        return -1;
    }
    if (a > b) {
        return 1;
    }
    // a必须等于
    return 0;
}

numbers.sort(compare);
```

这是因为JavaScript的sort方法接受compareFunction作为参数，然后sort会用它排序数组。在例子里，我们声明了一个用来比较数组元素的函数，使数组按升序排序。

1. 自定义排序

我们可以对任何对象类型的数组排序，也可以创建compareFunction来比较元素。例如，对象Person有名字和年龄属性，我们希望根据年龄排序，就可以这么写：

```
var friends = [
    {name: 'John', age: 30},
```

```
    {name: 'Ana', age: 20},
    {name: 'Chris', age: 25}
];

function comparePerson(a, b){
    if (a.age < b.age){
        return -1
    }
    if (a.age > b.age){
        return 1
    }
    return 0;
}

console.log(friends.sort(comparePerson));
```

在这个例子里，最后会输出`Ana(20)`, `Chris(25)`, `John(30)`。

2. 字符串排序

假如有这样一个数组：

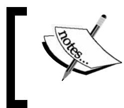
```
var names =['Ana', 'ana', 'john', 'John'];
console.log(names.sort());
```

你猜会输出什么？答案是这样的：

```
["Ana", "John", "ana", "john"]
```

既然a在字母表里排第一位，为何ana却排在了John之后呢？这是因为JavaScript在做字符比较的时候，是根据字符对应的ASCII值来比较的。例如，A、J、a、j对应的ASCII值分别是65、75、7、106。

虽然在字母表里a是最靠前的，但J的ASCII值比a的小，所以排在a前面。



想了解更多关于ASCII表的信息，请访问http://www.asciitable.com/。

现在，如果给`sort`传入一个忽略大小写的比较函数，将会输出`["Ana", "ana", "John", "john"]`：

```
names.sort(function(a, b){
    if (a.toLowerCase() < b.toLowerCase()){
        return -1
    }
    if (a.toLowerCase() > b.toLowerCase()){
        return 1
    }
    return 0;
});
```

假如对带有重音符号的字符做排序的话，我们可以用localCompare来实现：

```
var names2 = ['Maève', 'Maeve'];
console.log(names2.sort(function(a, b){
    return a.localeCompare(b);
}));
```

最后输出的结果将是["Maeve", "Maève"]。

3. 搜索

搜索有两个方法：indexOf方法返回与参数匹配的第一个元素的索引，lastIndexOf返回与参数匹配的最后一个元素的索引。我们来看看之前用过的numbers数组：

```
console.log(numbers.indexOf(10));
console.log(numbers.indexOf(100));
```

在这个示例中，第一行的输出是9，第二行的输出是–1（因为100不在数组里）。

下面的代码会返回同样的结果:

```
numbers.push(10);
console.log(numbers.lastIndexOf(10));
console.log(numbers.lastIndexOf(100));
```

我们往数组里加入了一个新的元素10，因此第二行会输出15（数组中的元素是1到15，还有10），第三行会输出–1（因为100不在数组里）。

2.5.4 输出数组为字符串

现在，我们学习最后两个方法：toString和join。

如果想把数组里所有元素输出为一个字符串，可以用toString方法：

```
console.log(numbers.toString());
```

1、2、3、4、5、6、7、8、9、10、11、12、13、14、15和10这些值都会在控制台中输出。

如果想用一个不同的分隔符（比如-）把元素隔开，可以用join方法：

```
var numbersString = numbers.join('-');
console.log(numbersString);
```

这将输出：

```
1-2-3-4-5-6-7-8-9-10-11-12-13-14-15-10
```

如果要把数组内容发送到服务器，或进行编码（知道了分隔符，解码也很容易），这会很有用。



有一些很棒的资源可以帮助你更深入地了解数组及其方法。

- 第一个是w3schools的数组页面：http://www.w3schools.com/js/js_arrays.asp。
- 第二个是w3schools的数组方法页面：http://www.w3schools.com/js/js_array_methods.asp。
- Mozilla的数组及其方法的页面也非常棒，还有不错的例子：https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array（http://goo.gl/vu1diT）
- 在JavaScript项目中使用数组时，也有一些很棒的类库。
  - Underscore：http://underscorejs.org/
  - Lo-Dash：http://lodash.com/

2.6　小结

在本章中，我们学习了最常用的数据结构：数组。我们学习了如何声明和初始化数组，给数组赋值，以及添加和移除数组元素，还学习了二维和多维数组以及数组的主要方法。这对我们在后面章节中编写自己的算法很有用。

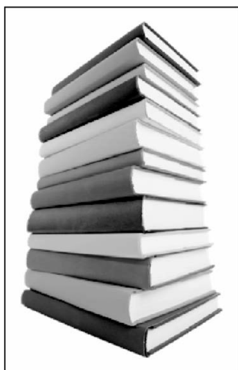
下一章，我们将学习栈，一种具有特殊行为的数组。

第 3 章 栈

数组是计算机科学中最常用的数据结构，上一章我们学习了如何创建和使用它。我们知道，可以在数组的任意位置上删除或添加元素。然而，有时候我们还需要一种在添加或删除元素时有更多控制的数据结构。有两种数据结构类似于数组，但在添加和删除元素时更为可控。它们就是栈和队列。本章我们主要讲述栈。

栈是一种遵从后进先出（LIFO）原则的有序集合。新添加的或待删除的元素都保存在栈的末尾，称作栈顶，另一端就叫栈底。在栈里，新元素都靠近栈顶，旧元素都接近栈底。

在现实生活中也能发现很多栈的例子。例如，下图里的一摞书或者餐厅里堆放的盘子。



栈也被用在编程语言的编译器和内存中保存变量、方法调用等。

3.1 栈的创建

我们将创建一个类来表示栈。让我们从基础开始，先声明这个类：

```
function Stack() {
    //各种属性和方法的声明
}
```

首先，我们需要一种数据结构来保存栈里的元素。可以选择数组：

```
var items = [];
```

接下来，要为我们的栈声明一些方法。

- push(element(s))：添加一个（或几个）新元素到栈顶。
- pop()：移除栈顶的元素，同时返回被移除的元素。
- peek()：返回栈顶的元素，不对栈做任何修改（这个方法不会移除栈顶的元素，仅仅返回它）。
- isEmpty()：如果栈里没有任何元素就返回true，否则返回false。
- clear()：移除栈里的所有元素。
- size()：返回栈里的元素个数。这个方法和数组的length属性很类似。

我们要实现的第一个方法是push。这个方法负责往栈里添加新元素，有一点很重要：该方法只添加元素到栈顶，也就是栈的末尾。push方法可以这样写：

```
this.push = function(element){
    items.push(element);
};
```

因为我们使用了数组来保存栈里的元素，所以可以用上一章里学到的数组的push方法来实现。

接着，我们来实现pop方法。这个方法主要用来移除栈里的元素。栈遵从LIFO原则，因此移出的是最后添加进去的元素。因此，我们可以用上一章讲数组时介绍的pop方法。栈的pop方法可以这样写：

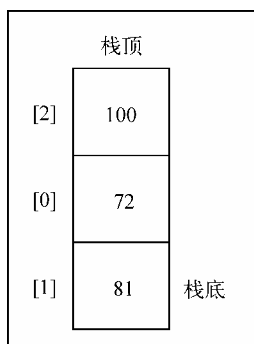
```
this.pop = function(){
    return items.pop();
};
```

只能用push和pop方法添加和删除栈中元素，这样一来，我们的栈自然就遵从了LIFO原则。

现在，为我们的类实现一些额外的辅助方法。如果想知道栈里最后添加的元素是什么，可以用peek方法。这个方法将返回栈顶的元素：

```
this.peek = function(){
    return items[items.length-1];
};
```

因为类内部是用数组保存元素的，所以访问数组的最后一个元素可以用 length - 1：



在上图中，有一个包含三个元素的栈，因此内部数组的长度就是3。数组中最后一项的位置是2，length - 1（3 – 1）正好是2。

下一个要实现的方法是 isEmpty，如果栈为空的话将返回true，否则就返回false：

```
this.isEmpty = function(){
    return items.length == 0;
};
```

使用isEmpty方法，我们能简单地判断内部数组的长度是否为0。

类似于数组的length属性，我们也能实现栈的length。对于集合，最好用size代替length。因为栈的内部使用数组保存元素，所以能简单地返回栈的长度：

```
this.size = function(){
    return items.length;
};
```

最后，我们来实现clear方法。clear方法用来移除栈里所有的元素，把栈清空。实现这个方法最简单的方式是：

```
this.clear = function(){
    items = [];
};
```

另外也可以多次调用pop方法，把数组中的元素全部移除，这样也能实现clear方法。

完成了！栈已经实现。通过一个例子来放松一下：为了检查栈里的内容，我们来实现一个辅助方法，叫print。它会把栈里的元素都输出到控制台：

```
this.print = function(){
    console.log(items.toString());
};
```

这样，我们就完整创建了栈！

栈的全部代码

实现栈之后，我们来看一看完整的代码：

```
function Stack() {

    var items = [];

    this.push = function(element){
        items.push(element);
    };

    this.pop = function(){
        return items.pop();
    };

    this.peek = function(){
        return items[items.length-1];
    };

    this.isEmpty = function(){
        return items.length == 0;
    };

    this.size = function(){
        return items.length;
    };

    this.clear = function(){
        items = [];
    };

    this.print = function(){
        console.log(items.toString());
    };
}
```

使用Stack类

在深入了解栈的应用前，我们先来学习如何使用Stack类。

首先，我们需要初始化Stack类。然后，验证一下栈是否为空（输出是true，因为还没有往栈里添加元素）。

```
var stack = new Stack();
console.log(stack.isEmpty()); //输出为true
```

接下来，往栈里添加一些元素（这里我们添加数字5和8；你可以添加任意类型的元素）：

```
stack.push(5);
stack.push(8);
```

如果调用peek方法，将会输出8，因为它是往栈里添加的最后一个元素：

```
console.log(stack.peek());//输出8
```

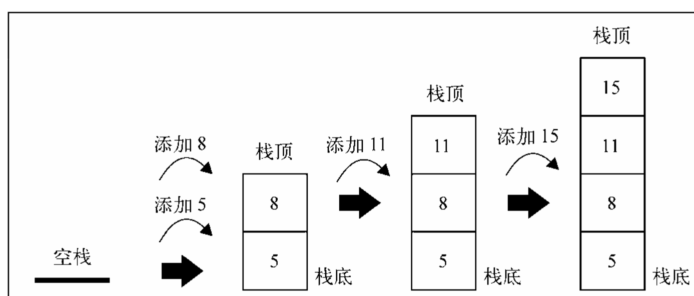
再添加一个元素：

```
stack.push(11);
console.log(stack.size()); //输出3
console.log(stack.isEmpty()); //输出false
```

我们往栈里添加了11。如果调用size方法，输出为3，因为栈里有三个元素（5、8和11）。如果我们调用isEmpty方法，会看到输出了false（因为栈里有三个元素，不是空栈）。最后，我们再添加一个元素：


```
stack.push(15);
```

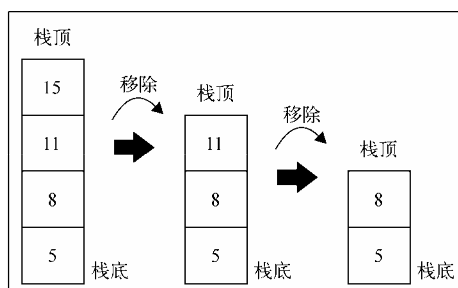
下图描绘了目前为止我们对栈的操作，以及栈的当前状态：



然后，调用两次pop方法从栈里移除2个元素：

```
stack.pop();
stack.pop();
console.log(stack.size()); //输出2
stack.print(); //输出[5, 8]
```

在两次调用pop方法前，我们的栈里有四个元素。调用两次后，现在栈里仅剩下5和8了。下图描绘这个过程的执行：

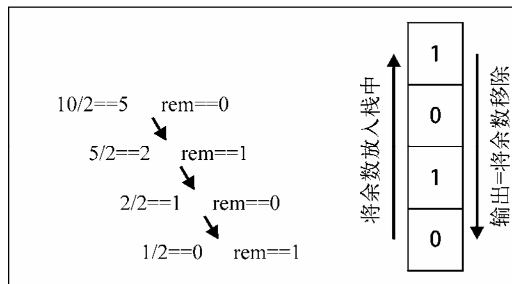


3.2 从十进制到二进制

我们已经学会了如何使用Stack类，现在就用它解决一些计算机科学中的问题。

现实生活中，我们主要使用十进制。但在计算科学中，二进制非常重要，因为计算机里的所有内容都是用二进制数字表示的（0和1）。没有十进制和二进制相互转化的能力，与计算机交流就很困难。

要把十进制转化成二进制，我们可以将该十进制数字和2整除（二进制是满二进一），直到结果是0为止。举个例子，把十进制的数字10转化成二进制的数字，过程大概是这样：



大学的计算机课一般都会先教这个进制转换。下面是对应的算法描述：

```
function divideBy2(decNumber){

    var remStack = new Stack(),
        rem,
        binaryString = '';

    while (decNumber > 0){ //{1}
        rem = Math.floor(decNumber % 2); //{2}
        remStack.push(rem); //{3}
        decNumber = Math.floor(decNumber / 2); //{4}
    }

    while (!remStack.isEmpty()){ //{5}
        binaryString += remStack.pop().toString();
    }

    return binaryString;
}
```

在这段代码里，当结果满足和2做整除的条件时（行{1}），我们会获得当前结果和2的余数，放到栈里（行{2}、{3}）。然后让结果和2做整除（行{4}）。另外请注意：JavaScript有数字类型，但是它不会区分究竟是整数还是浮点数。因此，要使用Math.floor函数让除法的操作仅返回整数部分。最后，用pop方法把栈中的元素都移除，把出栈的元素变成连接成字符串（行{5}）。

用刚才写的算法做一些测试，使用以下代码把结果输出到控制台里：

```
console.log(divideBy2(233)); //输出11101001
console.log(divideBy2(10)); //输出1010
console.log(divideBy2(1000)); //输出1111101000
```

我们很容易修改之前的算法，使之能把十进制转换成任何进制。除了让十进制数字和2整除转成二进制数，还可以传入其他任意进制的基数为参数，就像下面算法这样：

```
function baseConverter(decNumber, base){

    var remStack = new Stack(),
        rem,
        baseString = '',
        digits = '0123456789ABCDEF'; //{6}

    while (decNumber > 0){
        rem = Math.floor(decNumber % base);
        remStack.push(rem);
        decNumber = Math.floor(decNumber / base);
    }

    while (!remStack.isEmpty()){
        baseString += digits[remStack.pop()]; //{7}
    }

    return baseString;
}
```

3

我们只需要改变一个地方。在将十进制转成二进制时，余数是0或1；在将十进制转成八进制时，余数是0到8之间的数；但是将十进制转成16进制时，余数是0到8之间的数字加上A、B、C、D、E和F（对应10、11、12、13、14和15）。因此，我们需要对栈中的数字做个转化才可以（行{6}和行{7}）。

可以使用之前的算法，输出结果如下：

```
console.log(baseConverter(100345, 2)); //输出11000011111111001
console.log(baseConverter(100345, 8)); //输出303771
console.log(baseConverter(100345, 16)); //输出187F9
```

> 请在网上下载本书的代码，里面还有一些栈的应用实例，如平衡圆括号和**汉诺塔**。

3.3 小结

通过本章，我们学习了栈这一数据结构的相关知识。我们用代码自己实现了栈，还讲解了如何用push和pop往栈里添加和移除元素。另外还用进制转换这个例子讲解了如何使用栈。

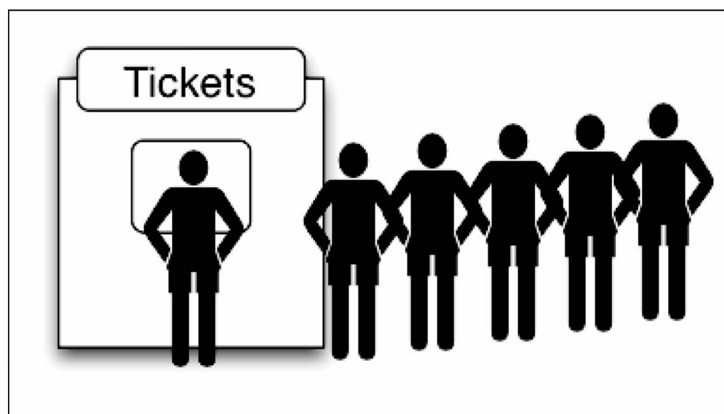
下一章将要学习队列。它和栈有很多相似之处，但有个重要区别，队列里的元素不遵循后进先出原则。

第 4 章 队　　列

我们已经学习了栈。队列和栈非常类似，但是使用了不同的原则，而非后进先出。你将在这一章学习这些内容。

队列是遵循FIFO（First In First Out，先进先出，也称为先来先服务）原则的一组有序的项。队列在尾部添加新元素，并从顶部移除元素。最新添加的元素必须排在队列的末尾。

在现实中，最常见的队列的例子就是排队：



还有，在电影院、自助餐厅、杂货店收银台，我们也都会排队。排在第一位的人会先接受服务。

在计算机科学中，一个常见的例子就是打印队列。比如说我们需要打印五份文档。我们会打开每个文档，然后点击打印按钮。每个文档都会被发送至打印队列。第一个发送到打印队列的文档会首先被打印，以此类推，直到打印完所有文档。

4.1　创建队列

我们需要创建自己的类来表示一个队列。先从最基本的声明类开始：

```
function Queue() {
    //这里是属性和方法
}
```

首先需要一个用于存储队列中元素的数据结构。我们可以使用数组，就像在上一章Stack类中那样使用（你会发现Queue类和Stack类非常类似，只是添加和移除元素的原则不同）：

```
var items = [];
```

接下来需要声明一些队列可用的方法。

- enqueue(element(s))：向队列尾部添加一个（或多个）新的项。
- dequeue()：移除队列的第一（即排在队列最前面的）项，并返回被移除的元素。
- front()：返回队列中第一个元素——最先被添加，也将是最先被移除的元素。队列不做任何变动（不移除元素，只返回元素信息——与Stack类的peek方法非常类似）。
- isEmpty()：如果队列中不包含任何元素，返回true，否则返回false。
- size()：返回队列包含的元素个数，与数组的length属性类似。

首先要实现的是enqueue方法。这个方法负责向队列添加新元素。这里有一个非常重要的细节，新的项只能添加到队列末尾：

```
this.enqueue = function(element){
    items.push(element);
};
```

既然我们使用数组来存储队列的元素，就可以用第2章和第3章中介绍过的JavaScript的array类的push方法。

接下来要实现dequeue方法。这个方法负责从队列移除项。由于队列遵循先进先出原则，最先添加的项也是最先被移除的。可以用第2章中介绍过的JavaScript的array类的shift方法。如果你不记得了，这里帮你回忆一下，shift方法会从数组中移除存储在索引0（第一个位置）的元素：

```
this.dequeue = function(){
    return items.shift();
};
```

只有enqueue方法和dequeue方法可以添加和移除元素，这样就确保了Queue类遵循先进先出原则。

现在来为我们的类实现一些额外的辅助方法。如果想知道队列最前面的项是什么，可以用front方法。这个方法会返回队列最前面的项（数组的索引为0）：

```
this.front = function(){
    return items[0];
};
```

下一个是isEmpty方法。如果队列为空，它会返回true，否则返回false（注意这个方法和Stack类里的一样）：

```
this.isEmpty = function(){
    return items.length == 0;
};
```

对于isEmpty方法，可以简单地验证内部数组的length是否为0。

我们也可以为Queue类实现类似于array类的length属性的方法。size方法也跟Stack类里的一样：

```
this.size = function(){
    return items.length;
};
```

完成！我们的Queue类就实现好了。也可以像Stack类一样增加一个print方法：

```
this.print = function(){
   console.log(items.toString());
};
```

现在我们真的完成了！

4.1.1　完整的Queue类

看看Queue类完整的实现是什么样的：

```
function Queue() {

    var items = [];

    this.enqueue = function(element){
        items.push(element);
    };

    this.dequeue = function(){
        return items.shift();
    };

    this.front = function(){
        return items[0];
    };

    this.isEmpty = function(){
        return items.length == 0;
    };

    this.clear = function(){
        items = [];
    };

    this.size = function(){
        return items.length;
    };
```

```
    this.print = function(){
        console.log(items.toString());
    };
}
```

> Queue类和Stack类非常类似。唯一的区别是dequeue方法和front方法，这是由于先进先出和后进先出原则的不同所造成的。

4.1.2 使用Queue类

首先要做的是实例化我们刚刚创建的Queue类，然后就可以验证它为空（输出为true，因为我们还没有向队列添加任何元素）：

```
var queue = new Queue();
console.log(queue.isEmpty()); //输出true
```

接下来，添加一些元素（添加"John"和"Jack"两个元素——你可以向队列添加任何类型的元素）：

```
queue.enqueue("John");
queue.enqueue("Jack");
```

添加另一个元素：

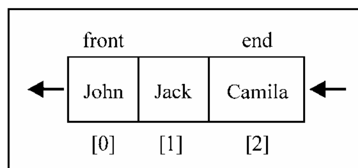
```
queue.enqueue("Camila");
```

再执行一些其他的命令：

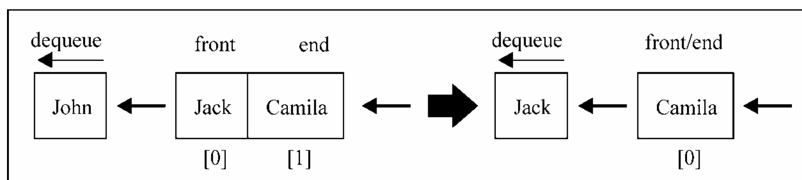
```
queue.print();
console.log(queue.size()); //输出3
console.log(queue.isEmpty()); //输出false
queue.dequeue();
queue.dequeue();
queue.print();
```

如果打印队列的内容，就会得到John、Jack和Camila这三个元素。因为我们向队列添加了三个元素，所以队列的大小为3（当然也就不为空了）。

下图展示了目前为止执行的所有入列操作，以及队列当前的状态：



然后，出列两个元素（执行两次dequeue方法）。下图展示了dequeue方法的执行过程：



最后，再次打印队列内容时，就只剩Camila一个元素了。前两个入列的元素出列了，最后入列的元素也将是最后出列的。也就是说，我们遵循了先进先出原则。

4.2　优先队列

队列大量应用在计算机科学以及我们的生活中，我们在之前话题中实现的默认队列也有一些修改版本。

其中一个修改版就是*优先队列*。元素的添加和移除是基于优先级的。一个现实的例子就是机场登机的顺序。头等舱和商务舱乘客的优先级要高于经济舱乘客。在有些国家，老年人和孕妇（或带小孩的妇女）登机时也享有高于其他乘客的优先级。

另一个现实中的例子是医院的（急诊科）候诊室。医生会优先处理病情比较严重的患者。通常，护士会鉴别分类，根据患者病情的严重程度放号。

实现一个优先队列，有两种选项：设置优先级，然后在正确的位置添加元素；或者用入列操作添加元素，然后按照优先级移除它们。在这个示例中，我们将会在正确的位置添加元素，因此可以对它们使用默认的出列操作：

```
function PriorityQueue() {

    var items = [];

    function QueueElement (element, priority){ // {1}
        this.element = element;
        this.priority = priority;
    }

    this.enqueue = function(element, priority){
        var queueElement = new QueueElement(element, priority);

        if (this.isEmpty()){
            items.push(queueElement); // {2}
        } else {
            var added = false;
            for (var i=0; i<items.length; i++){
                 if (queueElement.priority <
items[i].priority){
                    items.splice(i,0,queueElement); // {3}
                    added = true;
                    break; // {4}
```

```
                }
            }
            if (!added){ //{5}
                items.push(queueElement);
            }
        }
    };

//其他方法和默认的Queue实现相同
}
```

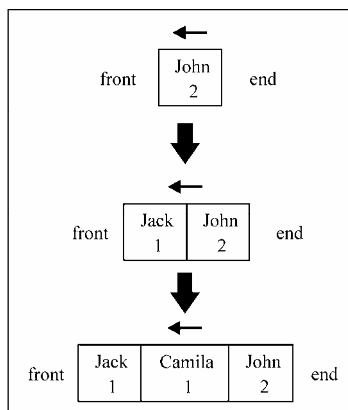
默认的Queue类和PriorityQueue类实现上的区别是，要向PriorityQueue添加元素，需要创建一个特殊的元素（行{1}）。这个元素包含了要添加到队列的元素（它可以是任意类型）及其在队列中的优先级。

如果队列为空，可以直接将元素入列（行{2}）。否则，就需要比较该元素与其他元素的优先级。当找到一个比要添加的元素的priority值更大（优先级更低）的项时，就把新元素插入到它之前（根据这个逻辑，对于其他优先级相同，但是先添加到队列的元素，我们同样遵循先进先出的原则）。要做到这一点，我们可以用第2章学习过的JavaScript的array类的splice方法。一旦找到priority值更大的元素，就插入新元素（行{3}）并终止队列循环（行{4}）。这样，队列也就根据优先级排序了。

如果要添加元素的priority值大于任何已有的元素，把它添加到队列的末尾就行了（行{5}）：

```
var priorityQueue = new PriorityQueue();
priorityQueue.enqueue("John", 2);
priorityQueue.enqueue("Jack", 1);
priorityQueue.enqueue("Camila", 1);
priorityQueue.print();
```

以上代码是一个使用PriorityQueue类的示例。在下图中可以看到每条命令的结果（以上代码的结果）：



第一个被添加的元素是优先级为2的John。因为此前队列为空，所以它是队列中唯一的元素。接下来，添加了优先级为1的Jack。由于Jack的优先级高于John，它就成了队列中的第一个元素。然后，添加了优先级也为1的Camila。Camila的优先级和Jack相同，所以它会被插入到Jack之后（因为Jack先被插入队列）；Camila的优先级高于John，所以它会被插入到John之前。

我们在这里实现的优先队列称为最小优先队列，因为优先级的值较小的元素被放置在队列最前面（1代表更高的优先级）。最大优先队列则与之相反，把优先级的值较大的元素放置在队列最前面。

4.3　循环队列——击鼓传花

还有另一个修改版的队列实现，就是*循环队列*。循环队列的一个例子就是击鼓传花游戏（Hot Potato）。在这个游戏中，孩子们围成一个圆圈，把花尽快地传递给旁边的人。某一时刻传花停止，这个时候花在谁手里，谁就退出圆圈结束游戏。重复这个过程，直到只剩一个孩子（胜者）。

在下面这个示例中，我们要实现一个模拟的击鼓传花游戏：

```
function hotPotato (nameList, num){

    var queue = new Queue(); // {1}

    for (var i=0; i<nameList.length; i++){
        queue.enqueue(nameList[i]); // {2}
    }

    var eliminated = '';
    while (queue.size() > 1){
        for (var i=0; i<num; i++){
            queue.enqueue(queue.dequeue()); // {3}
        }
        eliminated = queue.dequeue();// {4}
        console.log(eliminated + '在击鼓传花游戏中被淘汰。');
    }

    return queue.dequeue();// {5}
}

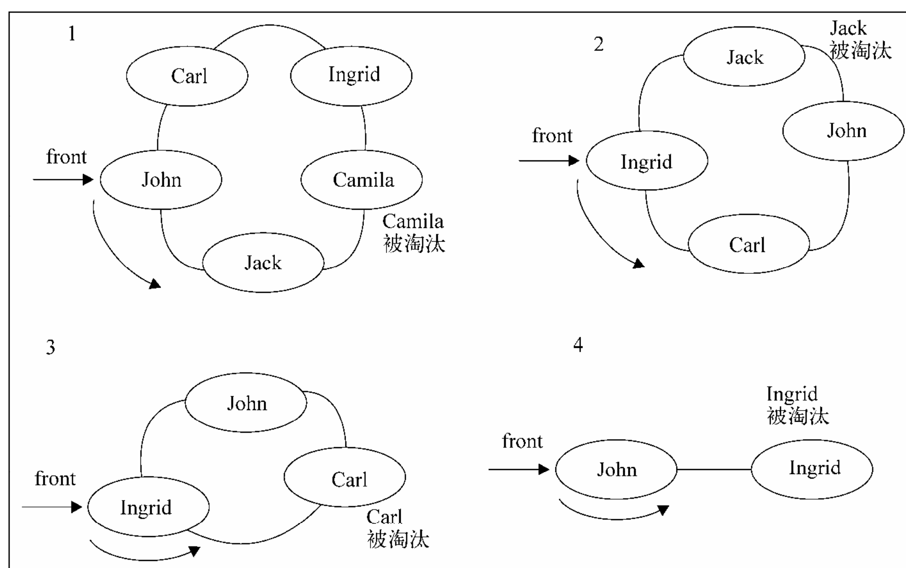
var names = ['John','Jack','Camila','Ingrid','Carl'];
var winner = hotPotato(names, 7);
console.log('胜利者：' + winner);
```

实现一个模拟的击鼓传花游戏，要用到这一章开头实现的Queue类（行{1}）。我们会得到一份名单，把里面的名字全都加入队列（行{2}）。给定一个数字，然后迭代队列。从队列开头移除一项，再将其添加到队列末尾（行{3}），模拟击鼓传花（如果你把花传给了旁边的人，你被淘汰的威胁立刻就解除了）。一旦传递次数达到给定的数字，拿着花的那个人就被淘汰了（从队列中移除——行{4}）。最后只剩下一个人的时候，这个人就是胜者（行{5}）。

以上算法的输出如下：

```
Camila在击鼓传花游戏中被淘汰。
Jack在击鼓传花游戏中被淘汰。
Carl在击鼓传花游戏中被淘汰。
Ingrid在击鼓传花游戏中被淘汰。
胜利者：John
```

下图模拟了这个输出过程：



你可以改变传入`hotPotato`函数的数字，模拟不同的场景。

4.4 小结

这一章我们学习了队列这种数据结构。我们实现了自己的队列算法，学习了如何通过`enqueue`方法和`dequeue`方法添加和移除元素。我们还学习了两种非常著名的特殊队列的实现：优先队列和循环队列（使用击鼓传花游戏的实现）。

在下一章中，我们将学习链表，一种比数组更复杂的数据结构。

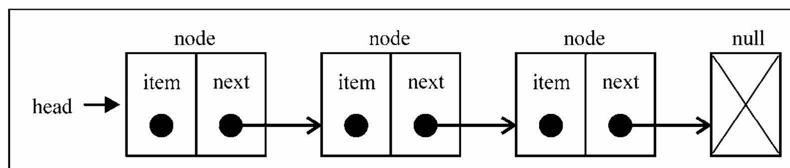
第 5 章

链　　表

我们在第2章中学习了数组这种数据结构。数组（或者也可以称为列表）是一种非常简单的存储数据序列的数据结构。在这一章中，你会学习如何实现和使用链表这种动态的数据结构，这意味着我们可以从中任意添加或移除项，它会按需进行扩容。

要存储多个元素，数组（或列表）可能是最常用的数据结构。正如本书之前提到过的，每种语言都实现了数组。这种数据结构非常方便，提供了一个便利的`[]`语法来访问它的元素。然而，这种数据结构有一个缺点：（在大多数语言中）数组的大小是固定的，从数组的起点或中间插入或移除项的成本很高，因为需要移动元素（尽管我们已经学过的JavaScript的`Array`类方法可以帮我们做这些事，但背后的情况同样是这样）。

链表存储有序的元素集合，但不同于数组，链表中的元素在内存中并不是连续放置的。每个元素由一个存储元素本身的节点和一个指向下一个元素的引用（也称指针或链接）组成。下图展示了一个链表的结构：



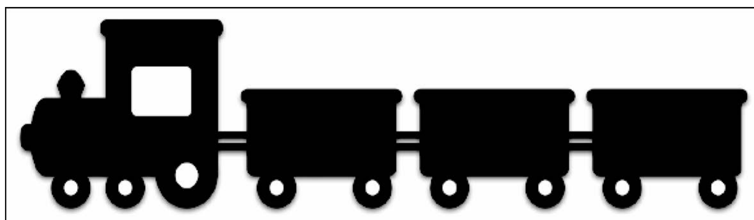
相对于传统的数组，链表的一个好处在于，添加或移除元素的时候不需要移动其他元素。然而，链表需要使用指针，因此实现链表时需要额外注意。数组的另一个细节是可以直接访问任何位置的任何元素，而要想访问链表中间的一个元素，需要从起点（表头）开始迭代列表直到找到所需的元素。

现实中也有一些链表的例子。第一个例子就是康加舞队。每个人是一个元素，手就是链向下一个人的指针。可以向队列中增加人——只需要找到想加入的点，断开连接，插入一个人，再重新连接起来。

另一个例子是寻宝游戏。你有一条线索，这条线索是指向寻找下一条线索的地点的指针。你

顺着这条链接去下一个地点,得到另一条指向再下一处的线索。得到列表中间的线索的唯一办法,就是从起点(第一条线索)顺着列表寻找。

还有一个可能是用来说明链表的最流行的例子,那就是火车。一列火车是由一系列车厢(也称车皮)组成的。每节车厢或车皮都相互连接。你很容易分离一节车皮,改变它的位置,添加或移除它。下图演示了一列火车。每节车皮都是列表的元素,车皮间的连接就是指针:



在这一章,我们会介绍链表和双向链表。但还是先从最简单的数据结构开始吧。

5.1　创建一个链表

理解了链表是什么之后,现在就要开始实现我们的数据结构了。以下是我们的LinkedList类的骨架:

```
function LinkedList() {

    var Node = function(element){ // {1}
        this.element = element;
        this.next = null;
    };

    var length = 0; // {2}
    var head = null; // {3}

    this.append = function(element){};
    this.insert = function(position, element){};
    this.removeAt = function(position){};
    this.remove = function(element){};
    this.indexOf = function(element){};
    this.isEmpty = function() {};
    this.size = function() {};
    this.toString = function(){};
    this.print = function(){};
}
```

LinkedList数据结构还需要一个Node辅助类(行{1})。Node类表示要加入列表的项。它包含一个element属性,即要添加到列表的值,以及一个next属性,即指向列表中下一个节点项的指针。

LinkedList类也有存储列表项的数量的length属性（内部/私有变量）（行{2}）。

另一个重要的点是，我们还需要存储第一个节点的引用。为此，可以把这个引用存储在一个称为head的变量中（行{3}）。

然后就是LinkedList类的方法。在实现这些方法之前，先来看看它们的职责。

- append(element)：向列表尾部添加一个新的项。
- insert(position, element)：向列表的特定位置插入一个新的项。
- remove(element)：从列表中移除一项。
- indexOf(element)：返回元素在列表中的索引。如果列表中没有该元素则返回-1。
- removeAt(position)：从列表的特定位置移除一项。
- isEmpty()：如果链表中不包含任何元素，返回true，如果链表长度大于0则返回false。
- size()：返回链表包含的元素个数。与数组的length属性类似。
- toString()：由于列表项使用了Node类，就需要重写继承自JavaScript对象默认的toString方法，让其只输出元素的值。

5.1.1 向链表尾部追加元素

向LinkedList对象尾部添加一个元素时，可能有两种场景：列表为空，添加的是第一个元素，或者列表不为空，向其追加元素。

下面是我们实现的append方法：

```
this.append = function(element){

    var node = new Node(element), //{1}
        current; //{2}

    if (head === null){ //列表中第一个节点 //{3}
        head = node;

    } else {
        current = head; //{4}

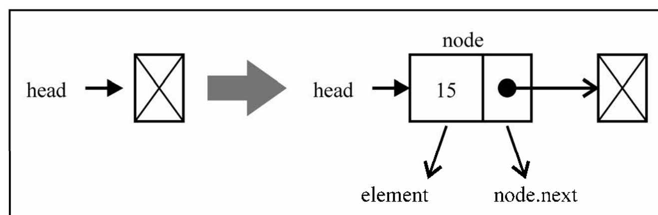
        //循环列表，直到找到最后一项
        while(current.next){
            current = current.next;
        }

        //找到最后一项，将其next赋为node，建立链接
        current.next = node; //{5}
    }

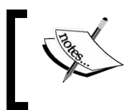
    length++; //更新列表的长度 //{6}
};
```

首先需要做的是把`element`作为值传入，创建`Node`项（行`{1}`）。

先来实现第一个场景：向为空的列表添加一个元素。当我们创建一个`LinkedList`对象时，`head`会指向`null`：



如果`head`元素为`null`（列表为空——行`{3}`），就意味着在向列表添加第一个元素。因此要做的就是让`head`元素指向`node`元素。下一个`node`元素将会自动成为`null`（请看5.1节的源代码）。

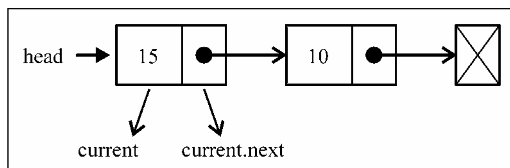


列表最后一个节点的下一个元素始终是`null`。

5

好了，我们已经说完了第一种场景。再来看看第二个，也就是向一个不为空的列表尾部添加元素。

要向列表的尾部添加一个元素，首先需要找到最后一个元素。记住，我们只有第一个元素的引用（行`{4}`），因此需要循环访问列表，直到找到最后一项。为此，我们需要一个指向列表中`current`项的变量（行`{2}`）。循环访问列表时，当`current.next`元素为`null`时，我们就知道已经到达列表尾部了。然后要做的就是让当前（也就是最后一个）元素的`next`指针指向想要添加到列表的节点（行`{5}`）。下图展示了这个行为：



而当一个`Node`元素被创建时，它的`next`指针总是`null`。这没问题，因为我们知道它会是列表的最后一项。

当然，别忘了递增列表的长度，这样就能控制它，轻松地得到列表的长度（行`{6}`）。

我们可以通过以下代码来使用和测试目前创建的数据结构：

```
var list = new LinkedList();
list.append(15);
list.append(10);
```

5.1.2　从链表中移除元素

现在，让我们看看如何从LinkedList对象中移除元素。移除元素也有两种场景：第一种是移除第一个元素，第二种是移除第一个以外的任一元素。我们要实现两种remove方法：第一种是从特定位置移除一个元素，第二种是根据元素的值移除元素（稍后我们会展示第二种remove方法）。

下面是根据给定位置移除一个元素的方法的实现：

```
this.removeAt = function(position){

    //检查越界值
    if (position > -1 && position < length){ // {1}
        var current = head, // {2}
            previous, // {3}
            index = 0; // {4}

        //移除第一项
        if (position === 0){ // {5}
            head = current.next;
        } else {

            while (index++ < position){ // {6}

                previous = current;     // {7}
                current = current.next; // {8}
            }

            //将previous与current的下一项链接起来：跳过current，从而移除它
            previous.next = current.next; // {9}
        }

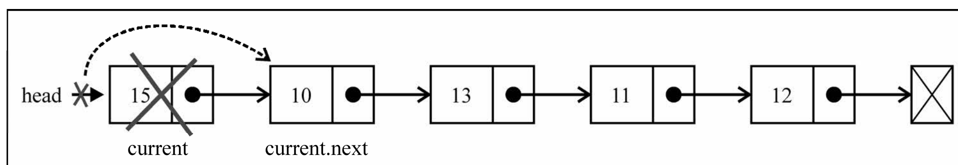
        length--; // {10}

        return current.element;

    } else {
        return null; // {11}
    }
};
```

一步一步来看这段代码。该方法要得到需要移除的元素的位置，就需要验证这个位置是有效的（行{1}）。从0（包括0）到列表的长度（size - 1，因为索引是从零开始的）都是有效的位置。如果不是有效的位置，就返回null（意即没有从列表中移除元素）。

一起来为第一种场景编写代码：我们要从列表中移除第一个元素（position === 0——行{5}）。下图展示了这个过程：



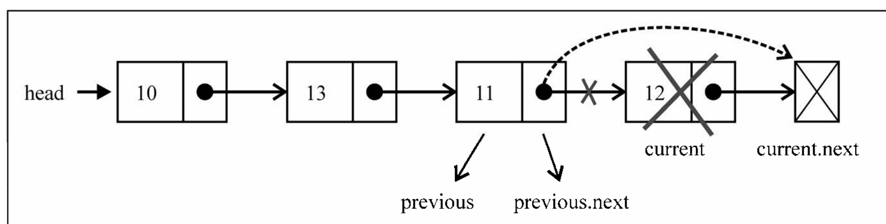
因此，如果想移除第一个元素，要做的就是让head指向列表的第二个元素。我们将用current变量创建一个对列表中第一个元素的引用（行{2}——我们还会用它来迭代列表，但稍等一下再说）。这样current变量就是对列表中第一个元素的引用。如果把head赋为current.next，就会移除第一个元素。

现在，假设我们要移除列表的最后一项或者中间某一项。为此，需要依靠一个细节来迭代列表，直到到达目标位置（行{6}——我们会使用一个用于内部控制和递增的index变量）：current变量总是为对所循环列表的当前元素的引用（行{8}）。我们还需要一个对当前元素的前一个元素的引用（行{7}）；它被命名为previous（行{3}）。

因此，要从列表中移除当前元素，要做的就是将previous.next和current.next链接起来（行{9}）。这样，当前元素就会被丢弃在计算机内存中，等着被垃圾回收器清除。

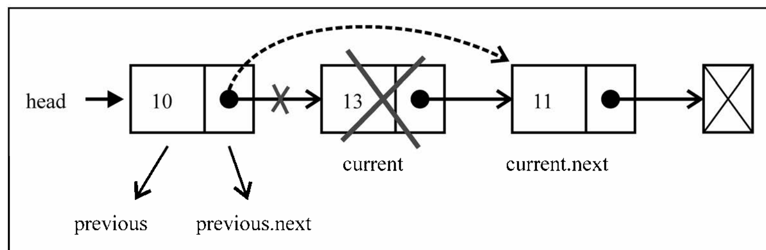
> 要更好地理解JavaScript垃圾回收器如何工作，请阅读https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management。

我们试着通过一些图表来更好地理解。首先考虑移除最后一个元素：



对于最后一个元素，当我们在行{6}跳出循环时，current变量将是对列表中最后一个元素的引用（要移除的元素）。current.next的值将是null（因为它是最后一个元素）。由于还保留了对previous元素的引用（当前元素的前一个元素），previous.next就指向了current。那么要移除current，要做的就是把previous.next的值改变为current.next。

现在来看看，对于列表中间的元素是否可以应用相同的逻辑：



current变量是对要移除元素的引用。previous变量是对要移除元素的前一个元素的引用。那么要移除current元素，需要做的就是将previous.next与current.next链接起来。因此，我们的逻辑对这两种情况都管用。

5.1.3　在任意位置插入一个元素

接下来，我们要实现insert方法。使用这个方法可以在任意位置插入一个元素。我们来看一看它的实现：

```
this.insert = function(position, element){

    //检查越界值
    if (position >= 0 && position <= length){ //{1}

        var node = new Node(element),
            current = head,
            previous,
            index = 0;

        if (position === 0){ //在第一个位置添加

            node.next = current; //{2}
            head = node;

        } else {
            while (index++ < position){ //{3}
                previous = current;
                current = current.next;
            }
            node.next = current; //{4}
            previous.next = node; //{5}
        }

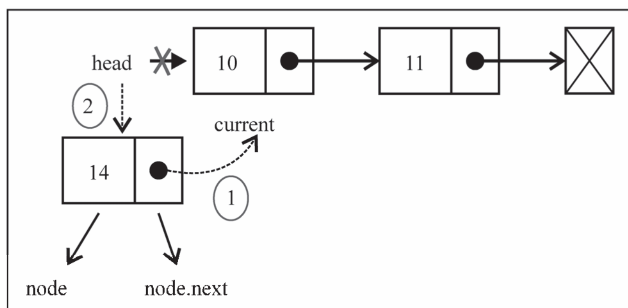
        length++; //更新列表的长度

        return true;

    } else {
        return false; //{6}
    }
};
```

由于我们处理的是位置，就需要检查越界值（行{1}，跟remove方法类似）。如果越界了，就返回false值，表示没有添加项到列表中（行{6}）。

现在我们要处理不同的场景。第一种场景，需要在列表的起点添加一个元素，也就是第一个位置。下图展示了这种场景：

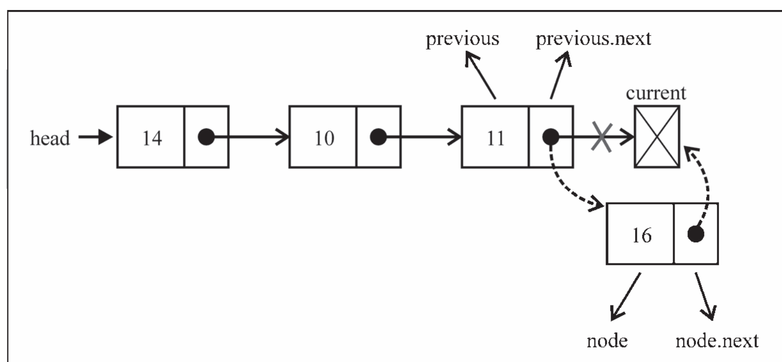


current变量是对列表中第一个元素的引用。我们需要做的是把node.next的值设为current（列表中第一个元素）。现在head和node.next都指向了current。接下来要做的就是把head的引用改为node（行{2}），这样列表中就有了一个新元素。

现在来处理第二种场景：在列表中间或尾部添加一个元素。首先，我们需要循环访问列表，找到目标位置（行{3}）。当跳出循环时，current变量将是对想要插入新元素的位置之后一个元素的引用，而previous将是对想要插入新元素的位置之前一个元素的引用。在这种情况下，我们要在previous和current之间添加新项。因此，首先需要把新项（node）和当前项链接起来（行{4}），然后需要改变previous和current之间的链接。我们还需要让previous.next指向node（行{5}）。

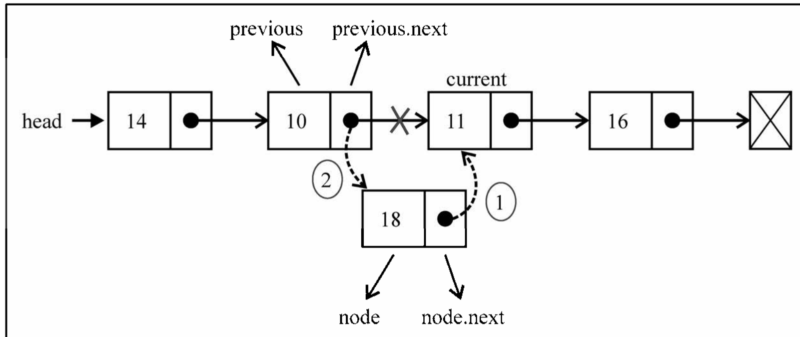
5

我们通过一张图表来看看代码所做的事：



如果我们试图向最后一个位置添加一个新元素，previous将是对列表最后一项的引用，而current将是null。在这种情况下，node.next将指向current，而previous.next将指向node，这样列表中就有了一个新的项。

现在来看看如何向列表中间添加一个新元素：



在这种情况下，我们试图将新的项（node）插入到previous和current元素之间。首先，我们需要把node.next的值指向current。然后把previous.next的值设为node。这样列表中就有了一个新的项。



> 使用变量引用我们需要控制的节点非常重要，这样就不会丢失节点之间的链接。我们可以只使用一个变量（previous），但那样会很难控制节点之间的链接。由于这个原因，最好是声明一个额外的变量来帮助我们处理这些引用。

5.1.4　实现其他方法

在这一节中，我们将会学习如何实现toString、indexOf、isEmpty和size等其他LinkedList类的方法。

1. toString方法

toString方法会把LinkedList对象转换成一个字符串。下面是toString方法的实现：

```
this.toString = function(){

    var current = head, //{1}
        string = '';    //{2}

    while (current) {   //{3}
        string = current.element; //{4}
        current = current.next;   //{5}
    }
    return string;                //{6}
};
```

首先，要循环访问列表中的所有元素，就需要有一个起点，也就是head。我们会把current变量当作索引（行{1}），控制循环访问列表。我们还需要初始化用于拼接元素值的变量（行{2}）。

接下来就是循环访问列表中的每个元素（行{3}）。我们要用current来检查元素是否存在（如果列表为空，或是到达列表中最后一个元素的下一位（null），while循环中的代码就不会执

行)。然后我们就得到了元素的内容,将其拼接到字符串中(行{4})。最后,继续迭代下一个元素(行{5})。

最后,返回列表内容的字符串(行{6})。

2. indexOf方法

indexOf是我们下一个要实现的方法。indexOf方法接收一个元素的值,如果在列表中找到它,就返回元素的位置,否则返回-1。

来看看它的实现:

```
this.indexOf = function(element){

    var current = head, //{1}
        index = -1;

    while (current) { //{2}
        if (element === current.element) {
            return index;       //{3}
        }
        index++;                //{4}
        current = current.next; //{5}
    }

    return -1;
};
```

一如既往,我们需要一个变量来帮助我们循环访问列表,这个变量是current,它的初始值是head(列表的第一个元素——我们还需要一个index变量来计算位置数(行{1}))。然后循环访问元素(行{2}),检查当前元素是否是我们要找的。如果是,就返回它的位置(行{3});如果不是,就继续计数(行{4}),检查列表中下一个节点(行{5})。

如果列表为空,或是到达列表的尾部(current = current.next将是null),循环就不会执行。如果没有找到值,就返回-1。

实现了这个方法,我们就可以实现remove等其他的方法:

```
this.remove = function(element){
    var index = this.indexOf(element);
    return this.removeAt(index);
};
```

我们已经有一个移除给定位置的一个元素的removeAt方法了。现在有了indexOf方法,如果传入元素的值,就能找到它的位置,然后调用removeAt方法并传入找到的位置。这样非常简单,如果需要更改removeAt方法的代码,这样也更容易——两个方法都会被更改(这就是重用代码的妙处)。这样,我们就不需要维护两个从列表中移除一项的方法,只需要一个!同时,removeAt方法将会检查边界约束。

3. isEmpty、size和getHead方法

isEmpty和size方法跟我们在上一章实现的一模一样。但我们还是来看一下：

```
this.isEmpty = function() {
    return length === 0;
};
```

如果列表中没有元素，isEmpty方法就返回true，否则返回false。

```
this.size = function() {
    return length;
};
```

size方法返回列表的length。和我们之前几章实现的类有所不同，列表的length是内部控制的，因为LinkedList是从头构建的。

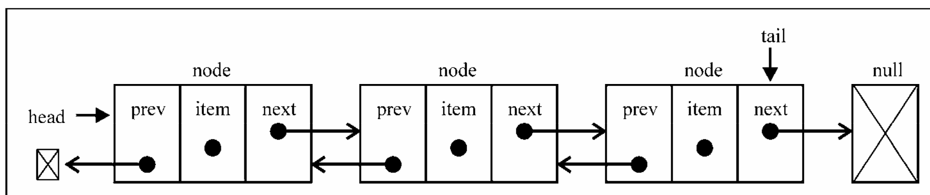
最后还有getHead方法：

```
this.getHead = function(){
    return head;
};
```

head变量是LinkedList类的私有变量（这意味着它不能在LinkedList实例外部被访问和更改，只有通过LinkedList实例才可以）。但是，如果我们需要在类的实现外部循环访问列表，就需要提供一种获取类的第一个元素的方法。

5.2　双向链表

链表有多种不同的类型，这一节介绍*双向链表*。双向链表和普通链表的区别在于，在链表中，一个节点只有链向下一个节点的链接，而在双向链表中，链接是双向的：一个链向下一个元素，另一个链向前一个元素，如下图所示：



先从实现DoublyLinkedList类所需的变动开始：

```
function DoublyLinkedList() {

    var Node = function(element){

        this.element = element;
        this.next = null;
        this.prev = null; //新增的
```

```
    };

    var length = 0;
    var head = null;
    var tail = null; //新增的

    //这里是方法
}
```

在代码中可以看到，LinkedList类和DoublyLinkedList类之间的区别标为新增的。在Node类里有prev属性（一个新指针），在DoublyLinkedList类里也有用来保存对列表最后一项的引用的tail属性。

双向链表提供了两种迭代列表的方法：从头到尾，或者反过来。我们也可以访问一个特定节点的下一个或前一个元素。在单向链表中，如果迭代列表时错过了要找的元素，就需要回到列表起点，重新开始迭代。这是双向链表的一个优点。

5.2.1 在任意位置插入一个新元素

向双向链表中插入一个新项跟（单向）链表非常类似。区别在于，链表只要控制一个next指针，而双向链表则要同时控制next和prev（previous，前一个）这两个指针。

这是向任意位置插入一个新元素的算法：

```
this.insert = function(position, element){

    //检查越界值
    if (position >= 0 && position <= length){

        var node = new Node(element),
            current = head,
            previous,
            index = 0;

        if (position === 0){ //在第一个位置添加

            if (!head){ //新增的 {1}
                head = node;
                tail = node;
            } else {
                node.next = current;
                current.prev = node; //新增的 {2}
                head = node;
            }
        } else  if (position === length) { //最后一项 //新增的

            current = tail; // {3}
            current.next = node;
            node.prev = current;
```

```
            tail = node;

        } else {
            while (index++ < position){ //{4}
                previous = current;
                current = current.next;
            }
            node.next = current; //{5}
            previous.next = node;

            current.prev = node; //新增的
            node.prev = previous; //新增的
        }

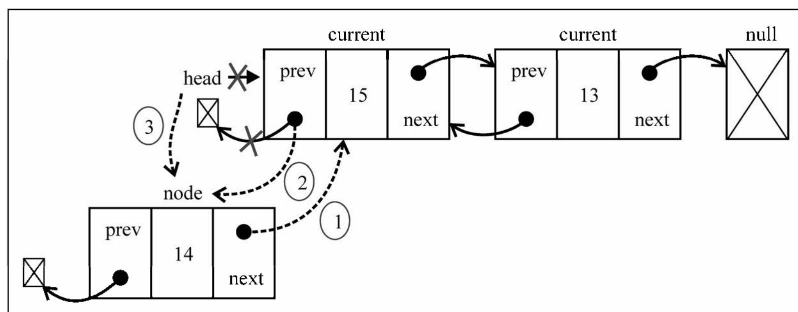
        length++; //更新列表的长度

        return true;

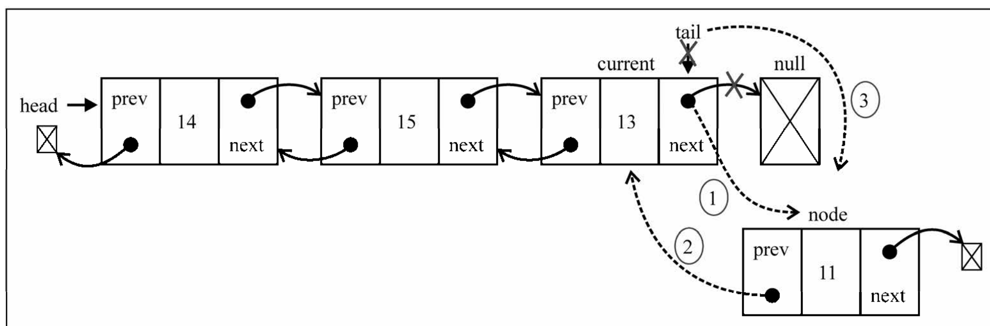
    } else {
        return false;
    }
};
```

我们来分析第一种场景：在列表的第一个位置（列表的起点）插入一个新元素。如果列表为空（行{1}），只需要把head和tail都指向这个新节点。如果不为空，current变量将是对列表中第一个元素的引用。就像我们在链表中所做的，把node.next设为current，而head将指向node（它将成为列表中的第一个元素）。不同之处在于，我们还需要为指向上一个元素的指针设一个值。current.prev指针将由指向null变为指向新元素（node——行{2}）。node.prev指针已经是null，因此不需要再更新任何东西。

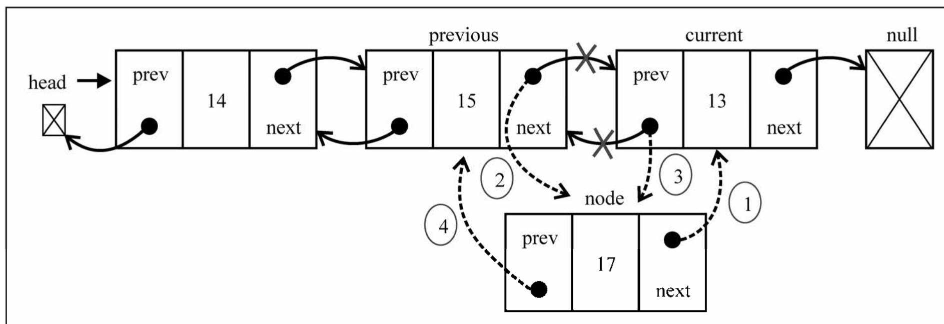
下图演示了这个过程：



现在来分析一下，假如我们要在列表最后添加一个新元素。这是一个特殊情况，因为我们还控制着指向最后一个元素的指针（tail）。current变量将引用最后一个元素（行{3}）。然后开始建立第一个链接：node.prev将引用current。current.next指针（指向null）将指向node（由于构造函数，node.next已经指向了null）。然后只剩一件事了，就是更新tail，它将由指向current变为指向node。下图展示了这些行为：



然后还有第三种场景：在列表中间插入一个新元素。就像我们在之前的方法中所做的，迭代列表，直到到达要找的位置（行{4}）。我们将在current和previous元素之间插入新元素。首先，node.next将指向current（行{5}），而previous.next将指向node，这样就不会丢失节点之间的链接。然后需要处理所有的链接：current.prev将指向node，而node.prev将指向previous。下图展示了这一过程：





> 我们可以对insert和remove这两个方法的实现做一些改进。在结果为否的情况下，我们可以把元素插入到列表的尾部。性能也可以有所改进，比如，如果position大于length/2，就最好从尾部开始迭代，而不是从头开始（这样就能迭代更少列表中的元素）。

5.2.2 从任意位置移除元素

从双向链表中移除元素跟链表非常类似。唯一的区别就是还需要设置前一个位置的指针。我们来看一下它的实现：

```
this.removeAt = function(position){

    //检查越界值
    if (position > -1 && position < length){
        var current = head,
```

```
            previous,
            index = 0;

        //移除第一项
        if (position === 0){

            head = current.next; // {1}

            //如果只有一项，更新tail //新增的
            if (length === 1){ // {2}
                tail = null;
            } else {
                head.prev = null; // {3}
            }

        } else if (position === length-1){ //最后一项 //新增的

            current = tail; // {4}
            tail = current.prev;
            tail.next = null;

        } else {

            while (index++ < position){ // {5}

                previous = current;
                current = current.next;
            }

            //将previous与current的下一项链接起来——跳过current
            previous.next = current.next; // {6}
            current.next.prev = previous; //新增的
        }

        length--;

        return current.element;

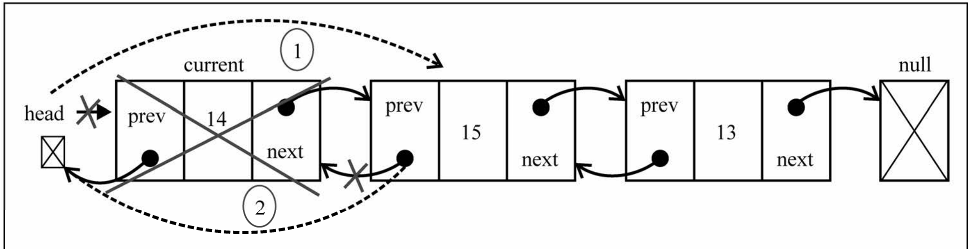
    } else {
        return null;
    }
};
```

我们需要处理三种场景：从头部、从中间和从尾部移除一个元素。

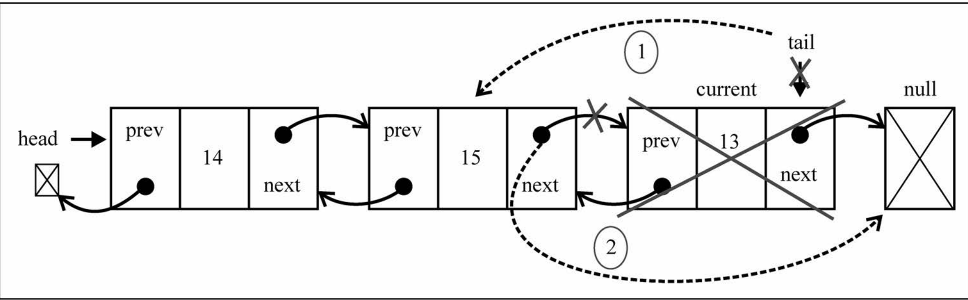
我们来看看如何移除第一个元素。`current`变量是对列表中第一个元素的引用，也就是我们想移除的元素。需要做的就是改变`head`的引用，将其从`current`改为下一个元素（`current.next`——行`{1}`）。但我们还需要更新`current.next`指向上一个元素的指针（因为第一个元素的`prev`指针是`null`）。因此，把`head.prev`的引用改为`null`（行`{3}`——因为`head`也指向列表中新的第一个元素，或者也可以用`current.next.prev`）。由于还需要控制`tail`的引用，我们可以检查要移除的元素是否是第一个元素，如果是，只需要把`tail`也设为`null`

（行{2}）。

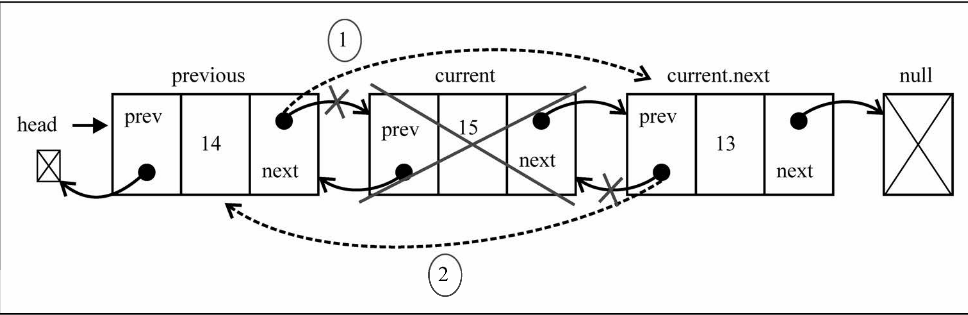
下图勾画了从双向链表移除第一个元素的过程：



下一种场景是从最后一个位置移除元素。既然已经有了对最后一个元素的引用（tail），我们就不需要为找到它而迭代列表。这样我们也就可以把tail的引用赋给current变量（行{4}）。接下来，需要把tail的引用更新为列表中倒数第二个元素（current.prev，或者tail.prev也可以）。既然tail指向了倒数第二个元素，我们就只需要把next指针更新为null（tail.next = null）。下图演示了这一行为：



第三种也是最后一种场景：从列表中间移除一个元素。首先需要迭代列表，直到到达要找的位置（行{5}）。current变量所引用的就是要移除的元素。那么要移除它，我们可以通过更新previous.next和current.next.prev的引用，在列表中跳过它。因此，previous.next将指向current.next，而current.next.prev将指向previous，如下图所示：

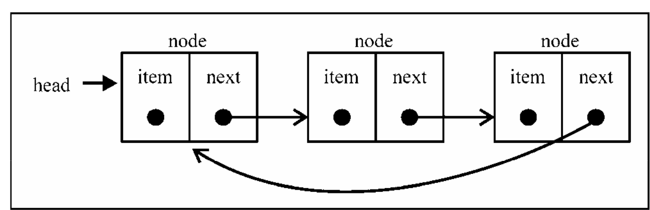




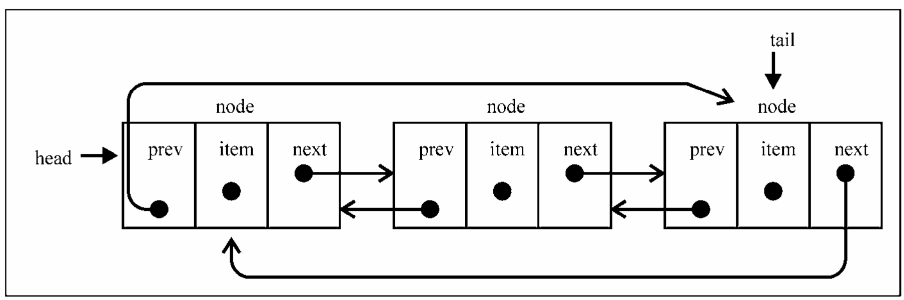
要了解双向链表的其他方法的实现，请参阅本书源代码。源代码的下载链接见本书前言。

5.3　循环链表

循环链表可以像链表一样只有单向引用，也可以像双向链表一样有双向引用。循环链表和链表之间唯一的区别在于，最后一个元素指向下一个元素的指针（`tail.next`）不是引用`null`，而是指向第一个元素（`head`），如下图所示。



双向循环链表有指向`head`元素的`tail.next`，和指向`tail`元素的`head.prev`。





我们并不打算在这本书中完整地介绍`CircularLinkedList`算法（源代码与`LinkedList`和`DoublyLinkedList`非常类似）。不过，你可以下载本书的源代码来访问这部分代码。

5.4　小结

在这一章中，你学习了链表这种数据结构，及其变体双向链表和循环链表。你学习了如何在任意位置添加和移除元素，以及如何循环访问链表。你还学习了链表相比数组最重要的优点，那就是无需移动链表中的元素，就能轻松地添加和移除元素。因此，当你需要添加和移除很多元素时，最好的选择就是链表，而非数组。

在下一章中，你将学习集合，这是我们要在本书中介绍的最后一种顺序数据结构。

第 6 章 集　合

6

迄今为止，我们已经学习了数组（列表）、栈、队列和链表（及其变种）等顺序数据结构。在这一章中，我们要学习集合这种数据结构。

集合是由一组无序且唯一（即不能重复）的项组成的。这个数据结构使用了与有限集合相同的数学概念，但应用在计算机科学的数据结构中。

在深入学习集合的计算机科学实现之前，我们先看看它的数学概念。在数学中，集合是一组不同的对象（的集）。

比如说，一个由大于或等于0的整数组成的自然数集合：$N = \{0, 1, 2, 3, 4, 5, 6, \cdots\}$。集合中的对象列表用“{}”（大括号）包围。

还有一个概念叫空集。空集就是不包含任何元素的集合。比如24和29之间的素数集合。由于24和29之间没有素数（除了1和自身，没有其他正因数的大于1的自然数），这个集合就是空集。空集用“{ }”表示。

你也可以把集合想象成一个既没有重复元素，也没有顺序概念的数组。

在数学中，集合也有并集、交集、差集等基本操作。在这一章中我们也会介绍这些操作。

6.1 创建一个集合

目前的JavaScript实现是基于2011年6月发布的ECMAScript 5.1（现代浏览器均已支持），它包括了我们在之前章节已经提到过的`Array`类的实现。ECMAScript 6（官方名称ECMAScript 2015，2015年6月发布）包括了`Set`类的实现。

> 你可以在 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Set（或http://goo.gl/2li2a5）看到ECMAScript 6的`Set`类的实现细节。

在这一章中，我们要实现的类就是以ECMAScript 6中`Set`类的实现为基础的。

以下是我们的Set类的骨架：

```
function Set() {
    var items = {};
}
```

有一个非常重要的细节，我们使用对象而不是数组来表示集合（items）。但也可以用数组实现。在这里我们用对象来实现，稍微有点儿不一样，也学习一下实现相似数据结构的新方法。同时，JavaScript的对象不允许一个键指向两个不同的属性，也保证了集合里的元素都是唯一的。

接下来，需要声明一些集合可用的方法（我们会尝试模拟与ECMAScript 6实现相同的Set类）。

- add(value)：向集合添加一个新的项。
- remove(value)：从集合移除一个值。
- has(value)：如果值在集合中，返回true，否则返回false。
- clear()：移除集合中的所有项。
- size()：返回集合所包含元素的数量。与数组的length属性类似。
- values()：返回一个包含集合中所有值的数组。

6.1.1　has(value)方法

首先要实现的是has(value)方法。这是因为它会被add、remove等其他方法调用。下面看看它的实现：

```
this.has = function(value){
    return value in items;
};
```

既然我们使用对象来存储集合的值，就可以用JavaScript的in操作符来验证给定的值是否是items对象的属性。

但这个方法还有更好的实现方式，如下：

```
this.has = function(value){
    return items.hasOwnProperty(value);
};
```

所有JavaScript对象都有hasOwnProperty方法。这个方法返回一个表明对象是否具有特定属性的布尔值。

6.1.2　add方法

接下来要实现add方法：

```
this.add = function(value){
    if (!this.has(value)){
        items[value] = value; //{1}
        return true;
    }
    return false;
};
```

对于给定的value，可以检查它是否存在于集合中。如果不存在，就把value添加到集合中（行{1}），返回true，表示添加了这个值。如果集合中已经有这个值，就返回false，表示没有添加它。



添加一个值的时候，把它同时作为键和值保存，因为这样有利于查找这个值。

6.1.3　remove和clear方法

下面要实现remove方法：

```
this.remove = function(value){
    if (this.has(value)){
        delete items[value]; //{2}
        return true;
    }
    return false;
};
```

在remove方法中，我们会验证给定的value是否存在于集合中。如果存在，就从集合中移除value（行{2}），返回true，表示值被移除；否则返回false。

既然用对象来存储集合的items对象，就可以简单地使用delete操作符从items对象中移除属性（行{2}）。

使用Set类的示例代码如下：

```
var set = new Set();
set.add(1);
set.add(2);
```



出于好奇，如果在执行以上代码之后，在控制台（console.log）输出items变量，谷歌Chrome就会输出如下内容：

```
Object {1: 1, 2: 2}
```

可以看到，这是一个有两个属性的对象。属性名就是添加到集合的值，同时它也是属性值。

如果想移除集合中的所有值，可以用clear方法：

```
this.clear = function(){
    items = {}; // {3}
};
```

要重置items对象，需要做的只是把一个空对象重新赋值给它（行{3}）。我们也可以迭代集合，用remove方法依次移除所有的值，但既然有更简单的方法，那样做就太麻烦了。

6.1.4　size方法

下一个要实现的是size方法（返回集合中有多少项）。这个方法有三种实现方式。

第一种方法是使用一个length变量，每当使用add或remove方法时控制它，就像在上一章中使用LinkedList类一样。

第二种方法，使用JavaScript内建的Object类的一个内建函数（ECMAScript 5以上版本）：

```
this.size = function(){
    return Object.keys(items).length; //{4}
};
```

JavaScript的Object类有一个keys方法，它返回一个包含给定对象所有属性的数组。在这种情况下，可以使用这个数组的length属性（行{4}）来返回items对象的属性个数。以上代码只能在现代浏览器中运行（比如IE9以上版本、Firefox 4以上版本、Chrome 5以上版本、Opera 12以上版本、Safari 5以上版本，等等）。

第三种方法是手动提取items对象的每一个属性，记录属性的个数并返回这个数字。这个方法可以在任何浏览器上运行，和之前的代码是等价的：

```
this.sizeLegacy = function(){
    var count = 0;
    for(var prop in items) { //{5}
        if(items.hasOwnProperty(prop)) //{6}
            ++count; //{7}
    }
    return count;
};
```

遍历items对象的所有属性（行{5}），检查它们是否是对象自身的属性（避免重复计数——行{6}）。如果是，就递增count变量的值（行{7}），最后在方法结束时返回这个数字。



> 不能简单地使用for-in语句遍历items对象的属性，递增count变量的值。还需要使用has方法（以验证items对象具有该属性），因为对象的原型包含了额外的属性（属性既有继承自JavaScript的Object类的，也有属于对象自身，未用于数据结构的）。

6.1.5 values方法

values方法也应用了相同的逻辑，提取items对象的所有属性，以数组的形式返回：

```
this.values = function(){
    return Object.keys(items);
};
```

以上代码只能在现代浏览器中运行。既然在本书中我们使用的测试浏览器是Chrome和Firefox，代码就能工作。

如果想让代码在任何浏览器中都能执行，可以用与之前代码等价的下面这段代码：

```
this.valuesLegacy = function(){
    var keys = [];
    for(var key in items){ //{7}
        keys.push(key); //{8}
    }
    return keys;
};
```

遍历items对象的所有属性（行{7}），把它们添加一个数组中（行{8}），并返回这个数组。

6.1.6 使用Set类

现在数据结构已经完成了，看看如何使用它吧。试着执行一些命令，测试我们的Set类：

```
var set = new Set();

set.add(1);
console.log(set.values()); //输出["1"]
console.log(set.has(1)); //输出true
console.log(set.size()); //输出1

set.add(2);
console.log(set.values()); //输出["1", "2"]
console.log(set.has(2)); //true
console.log(set.size()); //2

set.remove(1);
console.log(set.values()); //输出["2"]

set.remove(2);
console.log(set.values()); //输出[]
```

现在我们有了一个和ECMAScript 6中非常类似的Set类实现。如前所述，也可以用数组替代对象，存储元素。既然我们在第2章、第3章和第4章都用过数组，知道有不同方式实现同样的东西，这也不错。

6.2 集合操作

对集合可以进行如下操作。

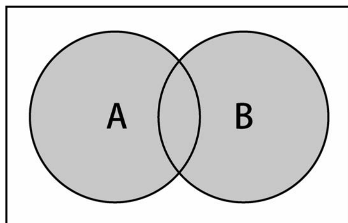
- 并集：对于给定的两个集合，返回一个包含两个集合中所有元素的新集合。
- 交集：对于给定的两个集合，返回一个包含两个集合中共有元素的新集合。
- 差集：对于给定的两个集合，返回一个包含所有存在于第一个集合且不存在于第二个集合的元素的新集合。
- 子集：验证一个给定集合是否是另一集合的子集。

6.2.1 并集

并集的数学概念，集合A和B的并集，表示为$A \cup B$，定义如下：

$$A \cup B = \{ x \mid x \in A \vee x \in B \}$$

意思是x（元素）存在于A中，或x存在于B中。下图展示了并集操作：



现在来实现`Set`类的`union`方法：

```
this.union = function(otherSet){
    var unionSet = new Set(); //{1}

    var values = this.values(); //{2}
    for (var i=0; i<values.length; i++){
        unionSet.add(values[i]);
    }

    values = otherSet.values(); //{3}
    for (var i=0; i<values.length; i++){
        unionSet.add(values[i]);
    }

    return unionSet;
};
```

首先需要创建一个新的集合，代表两个集合的并集（行`{1}`）。接下来，获取第一个集合（当前的`Set`类实例）所有的值（`values`），遍历并全部添加到代表并集的集合中（行`{2}`）。然后对第二个集合做同样的事（行`{3}`）。最后返回结果。

测试一下上面的代码：

```
var setA = new Set();
setA.add(1);
setA.add(2);
setA.add(3);

var setB = new Set();
setB.add(3);
setB.add(4);
setB.add(5);
setB.add(6);

var unionAB = setA.union(setB);
console.log(unionAB.values());
```

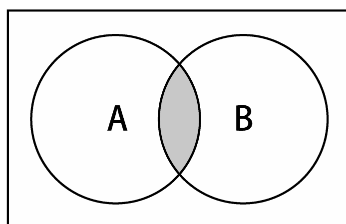
输出为`["1", "2", "3", "4", "5", "6"]`。注意元素3同时存在于*A*和*B*中，它在结果的集合中只出现一次。

6.2.2 交集

交集的数学概念，集合*A*和*B*的交集，表示为$A \cap B$，定义如下：

$$A \cap B = \{ x \mid x \in A \wedge x \in B \}$$

意思是*x*（元素）存在于*A*中，且*x*存在于*B*中。下图展示了交集操作：



现在来实现`Set`类的`intersection`方法：

```
this.intersection = function(otherSet){
    var intersectionSet = new Set(); //{1}

    var values = this.values();
    for (var i=0; i<values.length; i++){ //{2}
        if (otherSet.has(values[i])){    //{3}
            intersectionSet.add(values[i]); //{4}
        }
    }

    return intersectionSet;
}
```

intersection方法需要找到当前Set实例中，所有也存在于给定Set实例中的元素。首先创建一个新的Set实例，这样就能用它返回共有的元素（行{1}）。接下来，遍历当前Set实例所有的值（行{2}），验证它们是否也存在于otherSet实例（行{3}）。可以用这一章前面实现的has方法来验证元素是否存在于Set实例中。然后，如果这个值也存在于另一个Set实例中，就将其添加到创建的intersectionSet变量中（行{4}），最后返回它。

做些测试：

```
var setA = new Set();
setA.add(1);
setA.add(2);
setA.add(3);

var setB = new Set();
setB.add(2);
setB.add(3);
setB.add(4);

var intersectionAB = setA.intersection(setB);
console.log(intersectionAB.values());
```

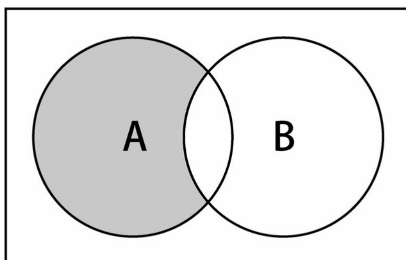
输出为["2", "3"]，因为2和3同时存在于两个集合中。

6.2.3　差集

差集的数学概念，集合A和B的差集，表示为$A-B$，定义如下：

$$A-B = \{ x \mid x \in A \wedge x \notin B \}$$

意思是x（元素）存在于A中，且x不存在于B中。下图展示了集合A和B的差集操作：



现在来实现Set类的difference方法：

```
this.difference = function(otherSet){
    var differenceSet = new Set(); //{1}

    var values = this.values();
    for (var i=0; i<values.length; i++){ //{2}
        if (!otherSet.has(values[i])){   //{3}
```

```
            differenceSet.add(values[i]); //{4}
        }
    }

    return differenceSet;
};
```

intersection方法会得到所有同时存在于两个集合中的值。而difference方法会得到所有存在于集合*A*但不存在于*B*的值。因此这两个方法在实现上唯一的区别就是行{3}。只获取不存在于otherSet实例中的值，而不是也存在于其中的值。行{1}、{2}和{4}是完全相同的。

（用跟intersection部分相同的集合）做些测试：

```
var setA = new Set();
setA.add(1);
setA.add(2);
setA.add(3);

var setB = new Set();
setB.add(2);
setB.add(3);
setB.add(4);

var differenceAB = setA.difference(setB);
console.log(differenceAB.values());
```

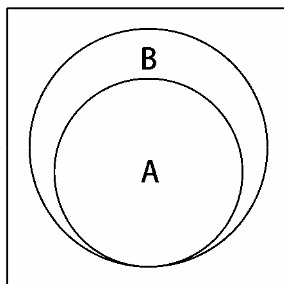
输出为["1"]，因为1是唯一一个仅存在于setA的元素。

6.2.4 子集

我们要介绍的最后一个集合操作是子集。子集的数学概念，集合*A*是*B*的子集（或集合*B*包含了*A*），表示为$A \subseteq B$，定义如下：

$$\forall x \{ x \in A \rightarrow x \in B \}$$

意思是集合*A*中的每一个*x*（元素），也需要存在于*B*中。下图展示了集合*A*是集合*B*的子集：



现在来实现Set类的subset方法：

```
this.subset = function(otherSet){
```

```
        if (this.size() > otherSet.size()){ //{1}
            return false;
        } else {
            var values = this.values();
            for (var i=0; i<values.length; i++){ //{2}
                if (!otherSet.has(values[i])){ //{3}
                    return false; //{4}
                }
            }
            return true; //{5}
        }
    };
```

首先需要验证的是当前Set实例的大小。如果当前实例中的元素比otherSet实例更多，它就不是一个子集（行{1}）。子集的元素个数需要小于或等于要比较的集合。

接下来要遍历集合中的所有元素（行{2}），验证这些元素也存在于otherSet中（行{3}）。如果有任何元素不存在于otherSet中，就意味着它不是一个子集，返回false（行{4}）。如果所有元素都存在于otherSet中，行{4}就不会被执行，那么就返回true（行{5}）。

检验一下上面的代码效果如何：

```
var setA = new Set();
setA.add(1);
setA.add(2);

var setB = new Set();
setB.add(1);
setB.add(2);
setB.add(3);

var setC = new Set();
setC.add(2);
setC.add(3);
setC.add(4);

console.log(setA.subset(setB));
console.log(setA.subset(setC));
```

我们有三个集合：setA是setB的子集（因此输出为true），然而setA不是setC的子集（setC只包含了setA中的2，而不包含1），因此输出为false。

6.3　小结

在这一章中，我们学习了如何从头实现一个与ECMAScript 6中定义的类似的Set类。我们还介绍了在其他编程语言的集合数据结构的实现中不常见的一些方法，比如并集、交集、差集和子集。因此，相比于其他编程语言目前的Set实现，我们实现了一个非常完备的Set类。

在下一章中，我们将会介绍散列和字典这两种非序列性的数据结构。

第 7 章 字典和散列表

在上一章中，我们学习了集合。本章我们会继续学习使用字典和散列表来存储唯一值（不重复的值）的数据结构。

集合、字典和散列表可以存储不重复的值。在集合中，我们感兴趣的是每个值本身，并把它当作主要元素。在字典中，我们用[键，值]的形式来存储数据。在散列表中也是一样（也是以[键，值]对的形式来存储数据）。但是两种数据结构的实现方式略有不同，本章中将会介绍。

7.1 字典

你已经知道，集合表示一组互不相同的元素（不重复的元素）。在字典中，存储的是[键，值]对，其中键名是用来查询特定元素的。字典和集合很相似，集合以[值，值]的形式存储元素，字典则是以[键，值]的形式来存储元素。字典也称作映射。

在本章中，我们会介绍几个在现实问题上使用字典数据结构的例子：一个实际的字典（单词和它们的释义）以及一个地址簿。

7.1.1 创建一个字典

与Set类相似，ECMAScript 6同样包含了一个Map类的实现，即我们所说的字典。

> 你可以在 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Map（或者 http://goo.gl/dm8VP6）找到ECMAScript 6中Map类实现的具体细节。

我们在本章中将要实现的类就是以ECMAScript 6中Map类的实现为基础的。你会发现它和Set类很相似（但不同于存储[值，值]对的形式，我们将要存储的是[键，值]对）。

这是我们的Dictionary类的骨架：

```
function Dictionary() {
    var items = {};
}
```

与Set类类似，我们将在一个Object的实例而不是数组中存储元素。

然后，我们需要声明一些映射/字典所能使用的方法。

- set(key,value)：向字典中添加新元素。
- remove(key)：通过使用键值来从字典中移除键值对应的数据值。
- has(key)：如果某个键值存在于这个字典中，则返回true，反之则返回false。
- get(key)：通过键值查找特定的数值并返回。
- clear()：将这个字典中的所有元素全部删除。
- size()：返回字典所包含元素的数量。与数组的length属性类似。
- keys()：将字典所包含的所有键名以数组形式返回。
- values()：将字典所包含的所有数值以数组形式返回。

1. has和set方法

我们首先来实现has(key)方法。之所以要先实现这个方法，是因为它会被set和remove等其他方法调用。我们可以通过如下代码来实现：

```
this.has = function(key) {
    return key in items;
}
```

这个方法的实现和我们之前在Set类中的实现是一样的。我们使用JavaScript中的in操作符来验证一个key是否是items对象的一个属性。

然后是set方法的实现：

```
this.set = function(key, value) {
    items[key] = value; //{1}
}
```

该方法接受一个key和一个value作为参数。我们直接将value设为items对象的key属性的值。它可以用来给字典添加一个新的值，或者用来更新一个已有的值。

2. remove方法

接下来，我们实现remove方法。它和Set类中的remove方法很相似，唯一的不同点在于我们将先搜索key（而不是value）：

```
this.remove = function(key) {
    if (this.has(key)) {
        delete items[key];
```

```
        return true;
    }
    return false;
}
```

然后我们可以使用JavaScript的remove操作符来从items对象中移除key属性。

3. get和values方法

如果我们想在字典中查找一个特定的项，并检索它的值，可以使用下面的方法：

```
this.get = function(key) {
    return this.has(key) ? items[key] : undefined;
};
```

get方法首先会验证我们想要检索的值是否存在（通过查找key值），如果存在，将返回该值，反之将返回一个undefined值（请记住undefined值和null值是不一样的，第1章中提到过这个概念）。

下一个是values方法。这个方法以数组的形式返回字典中所有values实例的值：

```
this.values = function() {
    var values = {};
    for (var k in items) { //{1}
        if (this.has(k)) {
            values.push(items[k]); //{2}
        }
    }
    return values;
};
```


首先，我们遍历items对象的所有属性值（行{1}）。为了确定值存在，我们使用has函数来验证key确实存在，然后将它的值加入values数组（行{2}）。最后，我们就能返回所有找到的值。



我们不能仅仅使用for-in语句来遍历items对象的所有属性，还需要使用has方法（验证items对象是否包含某个属性），因为对象的原型也会包含对象的其他属性（JavaScript基本的Object类中的属性将会被继承，并存在于当前对象中，而对于这个数据结构来说，我们并不需要它们）。

4. clear、size、keys和getItems方法

clear、size和keys方法与Set类中是完全一样的，因此我们就不在本章讨论了。

最后，我们来验证items属性的输出值。我们可以实现一个返回items变量的方法，叫作getItems：

```
this.getItems = function() {
    return items;
}
```

7.1.2　使用Dictionary类

首先，我们来创建一个Dictionary类的实例，然后给它添加三条电子邮件地址。我们将会使用这个dictionary实例来实现一个电子邮件地址簿。

使用我们创建的类来执行如下代码：

```
var dictionary = new Dictionary();
dictionary.set('Gandalf', 'gandalf@email.com');
dictionary.set('John', 'johnsnow@email.com');
dictionary.set('Tyrion', 'tyrion@email.com');
```

如果执行了如下代码，输出结果将会是true：

```
console.log(dictionary.has('Gandalf'));
```

下面的代码将会输出3，因为我们向字典实例中添加了三个元素：

```
console.log(dictionary.size());
```

现在，执行下面的几行代码：

```
console.log(dictionary.keys());
console.log(dictionary.values());
console.log(dictionary.get('Tyrion'));
```

输出结果分别如下所示：

```
["Gandalf", "John", "Tyrion"]
["gandalf@email.com", "johnsnow@email.com", "tyrion@email.com"]
tyrion@email.com
```

最后，再执行几行代码：

```
dictionary.remove('John');
```

再执行下面的代码：

```
console.log(dictionary.keys());
console.log(dictionary.values());
console.log(dictionary.getItems());
```

输出结果如下所示：

```
["Gandalf", "Tyrion"]
["gandalf@email.com", "tyrion@email.com"]
Object {Gandalf: "gandalf@email.com", Tyrion: "tyrion@email.com"}
```

移除了一个元素后，现在的dictionary实例中只包含两个元素了。加粗的一行表现了items对象的内部结构。

7.2 散列表

在本节中，你将会学到HashTable类，也叫HashMap类，是Dictionary类的一种散列表实现方式。

散列算法的作用是尽可能快地在数据结构中找到一个值。在之前的章节中，你已经知道如果要在数据结构中获得一个值（使用get方法），需要遍历整个数据结构来找到它。如果使用散列函数，就知道值的具体位置，因此能够快速检索到该值。散列函数的作用是给定一个键值，然后返回值在表中的地址。

举个例子，我们继续使用在前一节中使用的电子邮件地址簿。我们将要使用最常见的散列函数——“lose lose”散列函数，方法是简单地将每个键值中的每个字母的ASCII值相加。

名称/键	散列函数	散列值
Gandalf	71 + 97 + 110 + 100 + 97 + 108 + 102	685
John	74 + 111 + 104 + 110	399
Tyrion	84 + 121 + 114 +105 + 111 + 110	645

散列表	
[...]	
[399]	johnsnow@email.com
[...]	
[645]	tyrion@email.com
[...]	
[685]	gandalf@email.com
[...]	

7.2.1 创建一个散列表

我们将使用数组来表示我们的数据结构，该数据结构和上个话题中的图表所用的非常相似。

和之前一样，我们从搭建类的骨架开始：

```
function HashTable() {
    var table = [];
}
```

然后，给类添加一些方法。我们给每个类实现三个基础的方法。

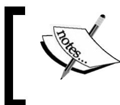
- put(key,value)：向散列表增加一个新的项（也能更新散列表）。
- remove(key)：根据键值从散列表中移除值。

- get(key)：返回根据键值检索到的特定的值。

在实现这三个方法之前，要实现的第一个方法是散列函数，它是HashTable类中的一个私有方法：

```
var loseloseHashCode = function (key) {
    var hash = 0;                          //{1}
    for (var i = 0; i < key.length; i++) { //{2}
        hash += key.charCodeAt(i);         //{3}
    }
    return hash % 37; //{4}
};
```

给定一个key参数，我们就能根据组成key的每个字符的ASCII码值的和得到一个数字。所以，首先需要一个变量来存储这个总和（行{1}）。然后，遍历key（行{2}）并将从ASCII表中查到的每个字符对应的ASCII值加到hash变量中（可以使用JavaScript的String类中的charCodeAt方法——行{3}）。最后，返回hash值。为了得到比较小的数值，我们会使用hash值和一个任意数做除法的余数（mod）。



要了解更多关于ASCII的信息，请访问http://www.asciitable.com/。

现在，有了散列函数，我们就可以实现put方法了：

```
this.put = function(key, value) {
    var position = loseloseHashCode(key); //{5}
    console.log(position + ' - ' + key); //{6}
    table[position] = value; //{7}
};
```

首先，根据给定的key，我们需要根据所创建的散列函数计算出它在表中的位置（行{5}）。为了便于展示信息，我们将计算出的位置输出至控制台（行{6}）。由于它不是必需的，我们也可以将这行代码移除。然后要做的，是将value参数添加到用散列函数计算出的对应的位置上。（行{7}）。

从HashTable实例中查找一个值也很简单。为此，将会实现一个get方法：

```
this.get = function (key) {
    return table[loseloseHashCode(key)];
};
```

首先，我们会使用所创建的散列函数来求出给定key所对应的位置。这个函数会返回值的位置，因此我们所要做的就是根据这个位置从数组table中获得这个值。

我们要实现的最后一个方法是remove方法：

```
this.remove = function(key) {
    table[loseloseHashCode(key)] = undefined;
};
```

要从HashTable实例中移除一个元素，只需要求出元素的位置（可以使用散列函数来获取）并赋值为undefined。

对于HashTable类来说，我们不需要像ArrayList类一样从table数组中将位置也移除。由于元素分布于整个数组范围内，一些位置会没有任何元素占据，并默认为undefined值。我们也不能将位置本身从数组中移除（这会改变其他元素的位置），否则，当下次需要获得或移除一个元素的时候，这个元素会不在我们用散列函数求出的位置上。

7.2.2 使用HashTable类

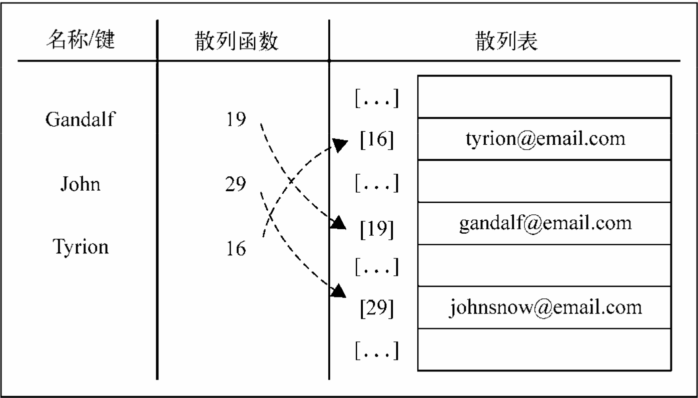
让我们执行一些代码来测试HashTable类：

```
var hash = new HashTable();
hash.put('Gandalf', 'gandalf@email.com');
hash.put('John', 'johnsnow@email.com');
hash.put('Tyrion', 'tyrion@email.com');
```

执行上述代码，会在控制台中获得如下输出：

```
19 - Gandalf
29 - John
16 - Tyrion
```

下面的图表展现了包含这三个元素的HashTable数据结构：



现在来测试get方法：

```
console.log(hash.get('Gandalf'));
console.log(hash.get('Loiane'));
```

获得如下的输出：

```
gandalf@email.com
undefined
```

由于Gandalf是一个在散列表中存在的键，get方法将会返回它的值。而由于Loiane是一个不存在的键，当我们试图在数组中根据位置获取值的时候（一个由散列函数生成的位置），返回值将会是undefined（即不存在）。

然后，我们试试从散列表中移除Gandalf：

```
hash.remove('Gandalf');
console.log(hash.get('Gandalf'));
```

由于Gandalf不再存在于表中，hash.get('Gandalf')方法将会在控制台上给出undefined的输出结果。

7.2.3　散列表和散列集合

散列表和散列映射是一样的，我们已经在本章中介绍了这种数据结构。

在一些编程语言中，还有一种叫作*散列集合*的实现。散列集合由一个集合构成，但是插入、移除或获取元素时，使用的是散列函数。我们可以重用本章中实现的所有代码来实现散列集合，不同之处在于，不再添加键值对，而是只插入值而没有键。例如，可以使用散列集合来存储所有的英语单词（不包括它们的定义）。和集合相似，散列集合只存储唯一的不重复的值。

7.2.4　处理散列表中的冲突

有时候，一些键会有相同的散列值。不同的值在散列表中对应相同位置的时候，我们称其为冲突。例如，我们看看下面的代码会得到怎样的输出结果：

```
var hash = new HashTable();
hash.put('Gandalf', 'gandalf@email.com');
hash.put('John', 'johnsnow@email.com');
hash.put('Tyrion', 'tyrion@email.com');
hash.put('Aaron', 'aaron@email.com');
hash.put('Donnie', 'donnie@email.com');
hash.put('Ana', 'ana@email.com');
hash.put('Jonathan', 'jonathan@email.com');
hash.put('Jamie', 'jamie@email.com');
hash.put('Sue', 'sue@email.com');
hash.put('Mindy', 'mindy@email.com');
hash.put('Paul', 'paul@email.com');
hash.put('Nathan', 'nathan@email.com');
```

输出结果如下：

```
19 - Gandalf
29 - John
16 - Tyrion
16 - Aaron
13 - Donnie
13 - Ana
5 - Jonathan
5 - Jamie
5 - Sue
32 - Mindy
32 - Paul
10 - Nathan
```

注意，Tyrion和Aaron有相同的散列值（16）。Donnie和Ana有相同的散列值（13），Jonathan、Jamie和Sue有相同的散列值（5），Mindy和Paul也有相同的散列值（32）。

那HashTable实例会怎样呢？执行之前的代码后散列表中会有哪些值呢？

为了获得结果，我们来实现一个叫作print的辅助方法，它会在控制台上输出HashTable中的值：

```
this.print = function() {
    for (var i = 0; i < table.length; ++i) { //{1}
        if (table[i] !== undefined) {        //{2}
            console.log(i + ": " + table[i]);//{3}
        }
    }
};
```

首先，遍历数组中的所有元素（行{1}）。当某个位置上有值的时候（行{2}），会在控制台上输出位置和对应的值（行{3}）。

现在来使用这个方法：

```
hash.print();
```

在控制台上得到如下的输出结果：

```
5: sue@email.com
10: nathan@email.com
13: ana@email.com
16: aaron@email.com
19: gandalf@email.com
29: johnsnow@email.com
32: paul@email.com
```

Jonathan、Jamie和Sue有相同的散列值，也就是5。由于Sue是最后一个被添加的，Sue将是在HashTable实例中占据位置5的元素。首先，Jonathan会占据这个位置，然后Jamie会覆盖它，然后Sue会再次覆盖。这对于其他发生冲突的元素来说也是一样的。

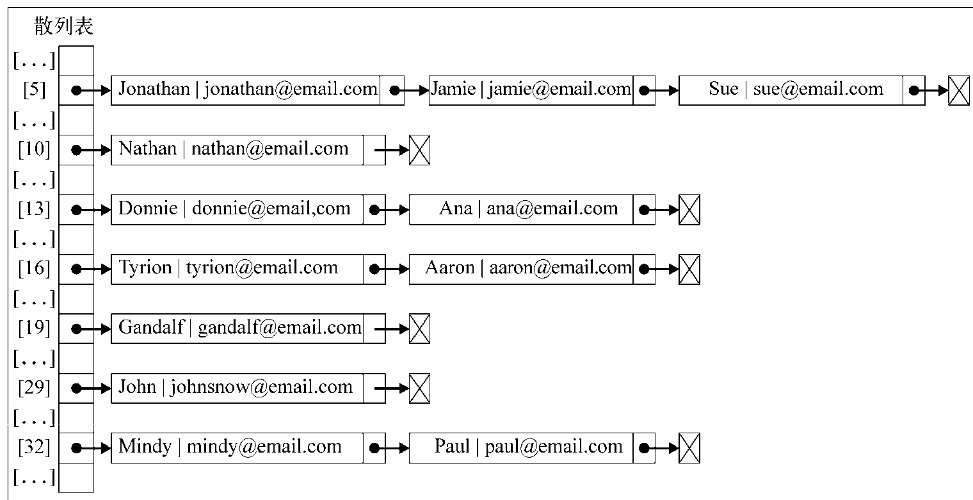
使用一个数据结构来保存数据的目的显然不是去丢失这些数据，而是通过某种方法将它们全

部保存起来。因此，当这种情况发生的时候就要去解决它。处理冲突有几种方法：分离链接、线性探查和双散列法。在本书中，我们会介绍前两种方法。

1. 分离链接

分离链接法包括为散列表的每一个位置创建一个链表并将元素存储在里面。它是解决冲突的最简单的方法，但是它在HashTable实例之外还需要额外的存储空间。

例如，我们在之前的测试代码中使用分离链接的话，输出结果将会是这样：



在位置5上，将会有包含三个元素的LinkedList实例；在位置13、16和32上，将会有包含两个元素的LinkedList实例；在位置10、19和29上，将会有包含单个元素的LinkedList实例。

对于分离链接和线性探查来说，只需要重写三个方法：put、get和remove。这三个方法在每种技术实现中都是不同的。

为了实现一个使用了分离链接的HashTable实例，我们需要一个新的辅助类来表示将要加入LinkedList实例的元素。我们管它叫ValuePair类（在HashTable类内部定义）：

```
var ValuePair = function(key, value){
    this.key = key;
    this.value = value;

    this.toString = function() {
        return '[' + this.key + ' - ' + this.value + ']';
    }
};
```

这个类只会将key和value存储在一个Object实例中。我们也重写了toString方法，以便之后在浏览器控制台中输出结果。

(1) put方法

我们来实现第一个方法，put方法，代码如下：

```
this.put = function(key, value){
    var position = loseloseHashCode(key);

    if (table[position] == undefined) { //{1}
        table[position] = new LinkedList();
    }
    table[position].append(new ValuePair(key, value)); //{2}
};
```

在这个方法中，将验证要加入新元素的位置是否已经被占据（行{1}）。如果这个位置是第一次被加入元素，我们会在这个位置上初始化一个LinkedList类的实例（你已经在第5章中学习过）。然后，使用第5章中实现的append方法向LinkedList实例中添加一个ValuePair实例（键和值）（行{2}）。

(2) get方法

然后，我们实现用来获取特定值的get方法：

```
this.get = function(key) {
    var position = loseloseHashCode(key);

    if (table[position] !== undefined){ //{3}

       //遍历链表来寻找键/值
       var current = table[position].getHead(); //{4}

       while(current.next){  //{5}
           if (current.element.key === key){ //{6}
               return current.element.value; //{7}
           }
           current = current.next; //{8}
       }

       //检查元素在链表第一个或最后一个节点的情况
       if (current.element.key === key){ //{9}
           return current.element.value;
       }
    }
    return undefined; //{10}
};
```

我们要做的第一个验证，是确定在特定的位置上是否有元素存在（行{3}）。如果没有，则返回一个undefined表示在HashTable实例中没有找到这个值（行{10}）。如果在这个位置上有值存在，我们知道这是一个LinkedList实例。现在要做的是遍历这个链表来寻找我们需要的元素。在遍历之前先要获取链表表头的引用（行{4}），然后就可以从链表的头部遍历到尾部（行{5}，current.next将会是null）。

Node链表包含next指针和element属性。而element属性又是ValuePair的实例，所以它又有value和key属性。可以通过current.element.next来获得Node链表的key属性，并通过比较它来确定它是否就是我们要找的键（行{6}）。（这就是要使用ValuePair这个辅助类来存储元素的原因。我们不能简单地存储值本身，这样就不能确定哪个值对应着特定的键。）如果key值相同，就返回Node的值（行{7}）；如果不相同，就继续遍历链表，访问下一个节点（行{8}）。

如果要找的元素是链表的第一个或最后一个节点，那么就不会进入while循环的内部。因此，需要在行{9}处理这种特殊的情况。

(3) remove方法

使用分离链接法从HashTable实例中移除一个元素和之前在本章实现的remove方法有一些不同。现在使用的是链表，我们需要从链表中移除一个元素。来看看remove方法的实现：

```
this.remove = function(key){
    var position = loseloseHashCode(key);

    if (table[position] !== undefined){

        var current = table[position].getHead();
        while(current.next){
            if (current.element.key === key){ //{11}
                table[position].remove(current.element); //{12}
                if (table[position].isEmpty()){ //{13}
                    table[position] = undefined; //{14}
                }
                return true; //{15}
            }
            current = current.next;
        }

        // 检查是否为第一个或最后一个元素
        if (current.element.key === key){ //{16}
            table[position].remove(current.element);
            if (table[position].isEmpty()){
                table[position] = undefined;
            }
            return true;
        }
    }

    return false; //{17}
};
```

在remove方法中，我们使用和get方法一样的步骤找到要找的元素。遍历LinkedList实例时，如果链表中的current元素就是要找的元素（行{11}），使用remove方法将其从链表中移除。然后进行一步额外的验证：如果链表为空了（行{13}——链表中不再有任何元素了），就将散列表这个位置的值设为undefined（行{14}），这样搜索一个元素或打印它的内容的时候，就

可以跳过这个位置了。最后，返回true表示这个元素已经被移除（行{15}）或者在最后返回false表示这个元素在散列表中不存在（行{17}）。同样，需要和get方法一样，处理元素在第一个或最后一个的情况（行{16}）。

重写了这三个方法后，我们就拥有了一个使用了分离链接法来处理冲突的HashMap实例。

2. 线性探查

另一种解决冲突的方法是线性探查。当想向表中某个位置加入一个新元素的时候，如果索引为index的位置已经被占据了，就尝试index+1的位置。如果index+1的位置也被占据了，就尝试index+2的位置，以此类推。

(1) put方法

让我们继续实现需要重写的三个方法。第一个是put方法：

```
this.put = function(key, value){
    var position = loseloseHashCode(key); // {1}

    if (table[position] == undefined) { // {2}
        table[position] = new ValuePair(key, value); // {3}
    } else {
        var index = ++position; // {4}
        while (table[index] != undefined){ // {5}
            index++; // {6}
        }
        table[index] = new ValuePair(key, value); // {7}
    }
};
```


和之前一样，先获得由散列函数生成的位置（行{1}），然后验证这个位置是否有元素存在（如果这个位置被占据了，将会通过行{2}的验证）。如果没有元素存在，就在这个位置加入新元素（行{3}——一个ValuePair的实例）。

如果这个位置已经被占据了，需要找到下一个没有被占据的位置（position的值是undefined），因此我们声明一个index变量并赋值为position+1（行{4}——在变量名前使用自增运算符++会先递增变量值然后再将其赋值给index）。然后验证这个位置是否被占据（行{5}），如果被占据了，继续将index递增（行{6}），直到找到一个没有被占据的位置。然后要做的，就是将值分配到这个位置（行{7}）。



在一些编程语言中，我们需要定义数组的大小。如果使用线性探查的话，需要注意的一个问题是数组的可用位置可能会被用完。在JavaScript中，我们不需要担心这个问题，因为我们不需要定义数组的大小，它可以根据需要自动改变大小——这是JavaScript内置的一个功能。

如果再次执行7.2.4节中插入数据的代码，下图展示使用了线性探查的散列表的最终结果：

散列表	
[...]	
[5]	Jonathan | jonathan@email.com
[6]	Jamie | jamie@email.com
[7]	Sue | sue@email.com
[...]	
[10]	Nathan | nathan@email.com
[...]	
[13]	Donnie | donnie@email.com
[14]	Ana | ana@email.com
[...]	
[16]	Tyrion | tyrion@email.com
[17]	Aaron | aaron@email.com
[18]	
[19]	Gandalf | gandalf@email.com
[...]	
[19]	John | johnsnow@email.com
[...]	

让我们来模拟一下散列表中的插入操作。

(1) 试着插入**Gandalf**。它的散列值是**19**，由于散列表刚刚被创建，位置**19**还是空的——可以在这里插入数据。

(2) 试着在位置**29**插入**John**。它也是空的，所以可以插入这个姓名。

(3) 试着在位置**16**插入**Tyrion**。它是空的，所以可以插入这个姓名。

(4) 试着插入**Aaron**，它的散列值也是**16**。位置**16**已经被**Tyrion**占据了，所以需要检查索引值为**position+1**的位置（**16+1**）。位置**17**是空的，所以可以在位置**17**插入**Aaron**。

(5) 接着，试着在位置**13**插入**Donnie**。它是空的，所以可以插入这个姓名。

(6) 想在位置**13**插入**Ana**，但是这个位置被占据了。因此在位置**14**进行尝试，它是空的，所以可以在这里插入姓名。

(7) 然后，在位置**5**插入**Jonathan**，这个位置是空的，所以可以插入这个姓名。

(8) 试着在位置**5**插入**Jamie**，但是这个位置被占了。所以跳至位置**6**，这个位置是空的，因此

可以在这个位置插入姓名。

(9) 试着在位置**5**插入**Sue**，但是位置被占据了。所以跳至位置**6**，但也被占了。接着跳至位置**7**，这里是空的，所以可以在这里插入姓名。

以此类推。

(2) get方法

现在插入了所有的元素，让我们实现get方法来获取它们的值吧：

```
this.get = function(key) {
    var position = loseloseHashCode(key);

    if (table[position] !== undefined){ //{8}
        if (table[position].key === key) { //{9}
            return table[position].value; //{10}
        } else {
            var index = ++position;
            while (table[index] === undefined
|| table[index].key !== key){ //{11}
                index++;
            }
            if (table[index].key === key) { //{12}
                return table[index].value; //{13}
            }
        }
    }
    return undefined; //{14}
};
```

要获得一个键对应的值，先要确定这个键存在（行{8}）。如果这个键不存在，说明要查找的值不在散列表中，因此可以返回undefined（行{14}）。如果这个键存在，需要检查我们要找的值是否就是这个位置上的值（行{9}）。如果是，就返回这个值（行{10}）。

如果不是，就在散列表中的下一个位置继续查找，直到找到一个键值与我们要找的键值相同的元素（行{11}）。然后，验证一下当前项就是我们要找的项（行{12}——只是为了确认一下）并且将它的值返回（行{13}）。

我们无法确定要找的元素实际上在哪个位置，这就是使用ValuePair来表示HashTable元素的原因。

(3) remove方法

remove方法和get方法基本相同，不同之处在于行{10}和{13}，它们将会由下面的代码代替：

```
table[index] = undefined;
```

要移除一个元素，只需要给其赋值为undefined，来表示这个位置不再被占据并且可以在必要时接受一个新元素。

7.2.5　创建更好的散列函数

我们实现的“lose lose”散列函数并不是一个表现良好的散列函数，因为它会产生太多的冲突。如果我们使用这个函数的话，会产生各种各样的冲突。一个表现良好的散列函数是由几个方面构成的：插入和检索元素的时间（即性能），当然也包括较低的冲突可能性。我们可以在网上找到一些不同的实现方法，或者也可以实现自己的散列函数。

另一个可以实现的比“lose lose”更好的散列函数是djb2：

```
var djb2HashCode = function (key) {
    var hash = 5381; //{1}
    for (var i = 0; i < key.length; i++) { //{2}
        hash = hash * 33 + key.charCodeAt(i); //{3}
    }
    return hash % 1013; //{4}
};
```

它包括初始化一个`hash`变量并赋值为一个质数（行`{1}`——大多数实现都使用`5381`），然后迭代参数`key`（行`{2}`），将`hash`与`33`相乘（用来当作一个魔力数），并和当前迭代到的字符的ASCII码值相加（行`{3}`）。

最后，我们将使用相加的和与另一个随机质数（比我们认为的散列表的大小要大——在本例中，我们认为散列表的大小为1000）相除的余数。

如果再次执行7.2.4节中插入数据的代码，这将是使用`djb2HashCode`代替`loseloseHashCode`的最终结果：

```
798 - Gandalf
838 - John
624 - Tyrion
215 - Aaron
278 - Donnie
925 - Ana
288 - Jonathan
962 - Jamie
502 - Sue
804 - Mindy
54 - Paul
223 - Nathan
```

没有冲突！

这并不是最好的散列函数，但这是最被社区推荐的散列函数之一。



也有一些为数字键值准备的散列函数，你可以在http://goo.gl/VtdN2x找到一系列的实现。

7.3　小结

在本章中，我们学习了字典的相关知识，了解了如何添加、移除和获取元素以及其他的一些方法。我们还了解了字典和集合的不同之处。

我们也学习了散列运算，怎样创建一个散列表（或者说散列映射）数据结构，如何添加、移除和获取元素，以及如何创建散列函数。我们学习了怎样使用两种不同的方法解决散列表中的冲突问题。

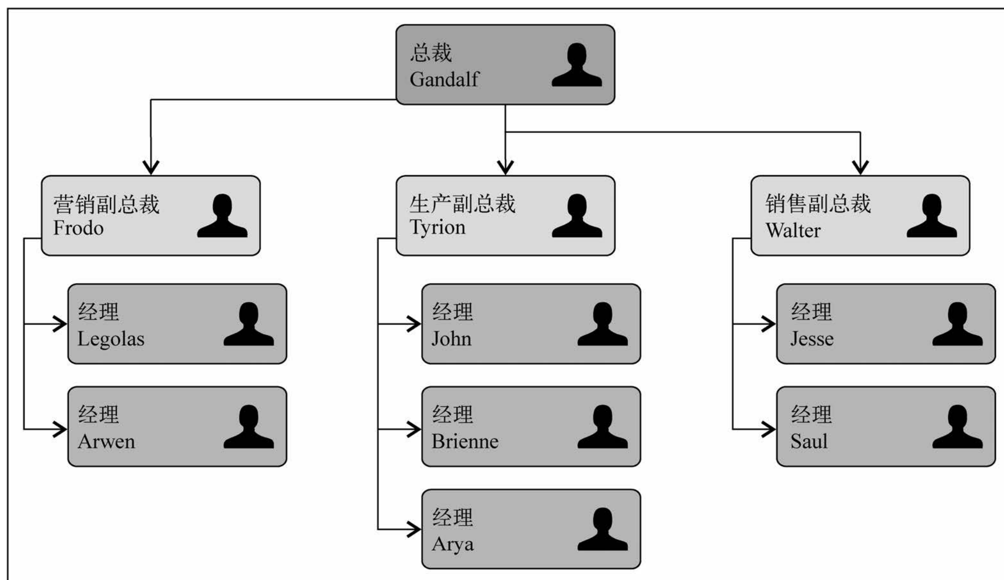
在下一章中，我们将学习一种新的数据结构——树。

第 8 章

树

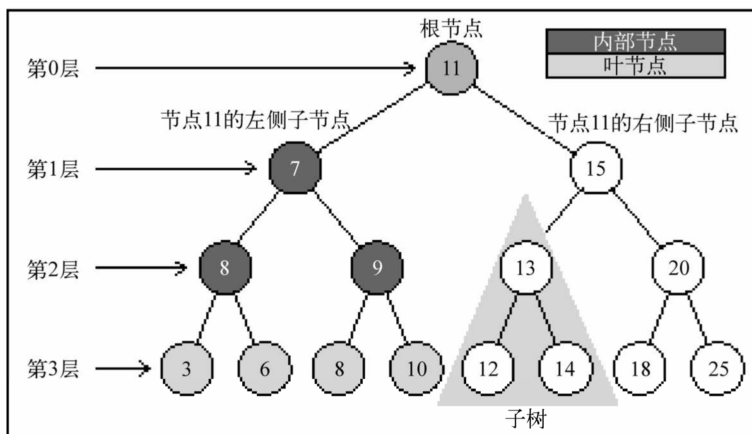
到目前为止，本书介绍了一些顺序数据结构，介绍的第一个非顺序数据结构是散列表。在本章，我们将要学习另一种非顺序数据结构——树，它对于存储需要快速查找的数据非常有用。

树是一种分层数据的抽象模型。现实生活中最常见的树的例子是家谱，或是公司的组织架构图，如下图所示：



8.1 树的相关术语

一个树结构包含一系列存在父子关系的节点。每个节点都有一个父节点（除了顶部的第一个节点）以及零个或多个子节点：



位于树顶部的节点叫作根节点（11）。它没有父节点。树中的每个元素都叫作节点，节点分为内部节点和外部节点。至少有一个子节点的节点称为内部节点（7、5、9、15、13和20是内部节点）。没有子元素的节点称为外部节点或叶节点（3、6、8、10、12、14、18和25是叶节点）。

一个节点可以有祖先和后代。一个节点（除了根节点）的祖先包括父节点、祖父节点、曾祖父节点等。一个节点的后代包括子节点、孙子节点、曾孙节点等。例如，节点5的祖先有节点7和节点11，后代有节点3和节点6。

有关树的另一个术语是子树。子树由节点和它的后代构成。例如，节点13、12和14构成了上图中树的一棵子树。

节点的一个属性是深度，节点的深度取决于它的祖先节点的数量。比如，节点3有3个祖先节点（5、7和11），它的深度为3。

树的高度取决于所有节点深度的最大值。一棵树也可以被分解成层级。根节点在第0层，它的子节点在第1层，以此类推。上图中的树的高度为3（最大高度已在图中表示——第3层）。

现在我们知道了与树相关的一些最重要的概念，下面来学习更多有关树的知识。

8.2　二叉树和二叉搜索树

二叉树中的节点最多只能有两个子节点：一个是左侧子节点，另一个是右侧子节点。这些定义有助于我们写出更高效的向/从树中插入、查找和删除节点的算法。二叉树在计算机科学中的应用非常广泛。

二叉搜索树（BST）是二叉树的一种，但是它只允许你在左侧节点存储（比父节点）小的值，在右侧节点存储（比父节点）大（或者等于）的值。上一节的图中就展现了一棵二叉搜索树。

二叉搜索树将是我们在本章中要研究的数据结构。

8.2.1　创建BinarySearchTree类

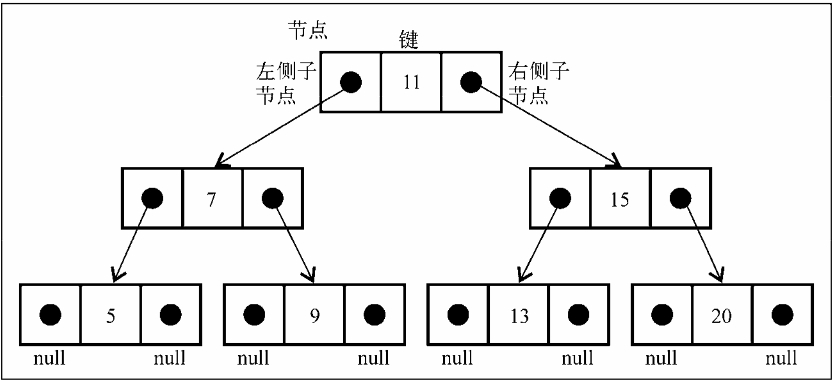
让我们开始创建自己的BinarySearchTree类。首先，声明它的结构：

```
function BinarySearchTree() {

   var Node = function(key){ //{1}
       this.key = key;
       this.left = null;
       this.right = null;
   };

   var root = null; //{2}
}
```

下图展现了二叉搜索树数据结构的组织方式：



和链表一样，将通过指针来表示节点之间的关系（术语称其为边）。在双向链表中，每个节点包含两个指针，一个指向下一个节点，另一个指向上一个节点。对于树，使用同样的方式（也使用两个指针）。但是，一个指向左侧子节点，另一个指向右侧子节点。因此，将声明一个Node类来表示树中的每个节点（行{1}）。值得注意的一个小细节是，不同于在之前的章节中将节点本身称作节点或项，我们将会称其为键。键是树相关的术语中对节点的称呼。

我们将会遵循和LinkedList类中相同的模式（第5章），这表示也将声明一个变量以控制此数据结构的第一个节点。在树中，它不再是头节点，而是根元素（行{2}）。

然后，我们需要实现一些方法。下面是将要在树类中实现的方法。

- insert(key)：向树中插入一个新的键。
- search(key)：在树中查找一个键，如果节点存在，则返回true；如果不存在，则返回false。
- inOrderTraverse：通过中序遍历方式遍历所有节点。
- preOrderTraverse：通过先序遍历方式遍历所有节点。

- ❑ postOrderTraverse：通过后序遍历方式遍历所有节点。
- ❑ min：返回树中最小的值/键。
- ❑ max：返回树中最大的值/键。
- ❑ remove(key)：从树中移除某个键。

我们将在后面的小节中实现每个方法。

8.2.2 向树中插入一个键

本章要实现的方法会比前几章实现的方法稍微复杂一些。我们将会在方法中使用很多递归。如果你对递归还不熟悉的话，请先参考11.1节。

下面的代码是用来向树插入一个新键的算法的第一部分：

```
this.insert = function(key){

    var newNode = new Node(key); //{1}

    if (root === null){ //{2}
        root = newNode;
    } else {
        insertNode(root,newNode); //{3}
    }
};
```

要向树中插入一个新的节点（或项），要经历三个步骤。

第一步是创建用来表示新节点的Node类实例（行{1}）。只需要向构造函数传递我们想用来插入树的节点值，它的左指针和右指针的值会由构造函数自动设置为null。

第二步要验证这个插入操作是否为一种特殊情况。这个特殊情况就是我们要插入的节点是树的第一个节点（行{2}）。如果是，就将根节点指向新节点。

第三步是将节点加在非根节点的其他位置。这种情况下，需要一个私有的辅助函数（行{3}），函数定义如下：

```
var insertNode = function(node, newNode){
    if (newNode.key < node.key){ //{4}
        if (node.left === null){   //{5}
            node.left = newNode;   //{6}
        } else {
            insertNode(node.left, newNode); //{7}
        }
    } else {
        if (node.right === null){  //{8}
            node.right = newNode;  //{9}
        } else {
            insertNode(node.right, newNode); //{10}
```

```
        }
    }
};
```

insertNode函数会帮助我们找到新节点应该插入的正确位置。下面是这个函数实现的步骤。

- 如果树非空，需要找到插入新节点的位置。因此，在调用insertNode方法时要通过参数传入树的根节点和要插入的节点。
- 如果新节点的键小于当前节点的键（现在，当前节点就是根节点）（行{4}），那么需要检查当前节点的左侧子节点。如果它没有左侧子节点（行{5}），就在那里插入新的节点。如果有左侧子节点，需要通过递归调用insertNode方法（行{7}）继续找到树的下一层。在这里，下次将要比较的节点将会是当前节点的左侧子节点。
- 如果节点的键比当前节点的键大，同时当前节点没有右侧子节点（行{8}），就在那里插入新的节点（行{9}）。如果有右侧子节点，同样需要递归调用insertNode方法，但是要用来和新节点比较的节点将会是右侧子节点。

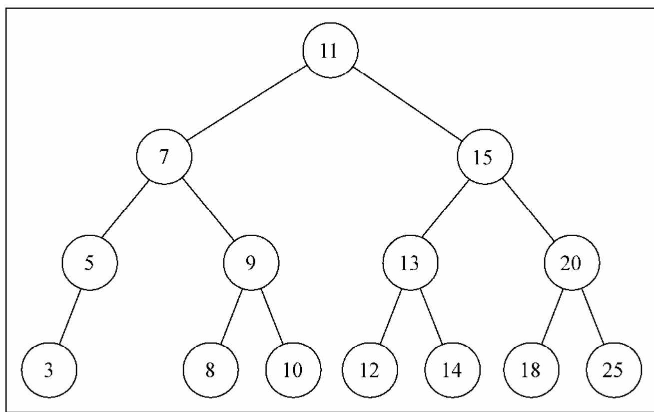
让我们通过一个例子来更好地理解这个过程。

考虑下面的情景：我们有一个新的树，并且想要向它插入第一个值。

```
var tree = new BinarySearchTree();
tree.insert(11);
```

这种情况下，树中有一个单独的节点，根指针将会指向它。源代码的行{2}将会执行。

现在，来考虑下图所示树结构的情况：



创建上图所示的树的代码如下，它们接着上面一段代码（插入了键为11的节点）之后输入执行：

```
tree.insert(7);
tree.insert(15);
tree.insert(5);
tree.insert(3);
```

```
tree.insert(9);
tree.insert(8);
tree.insert(10);
tree.insert(13);
tree.insert(12);
tree.insert(14);
tree.insert(20);
tree.insert(18);
tree.insert(25);
```

同时我们想要插入一个值为6的键，执行下面的代码：

```
tree.insert(6);
```

下面的步骤将会被执行。

(1) 树不是空的，行{3}的代码将会执行。insertNode方法将会被调用（root, key[6]）。

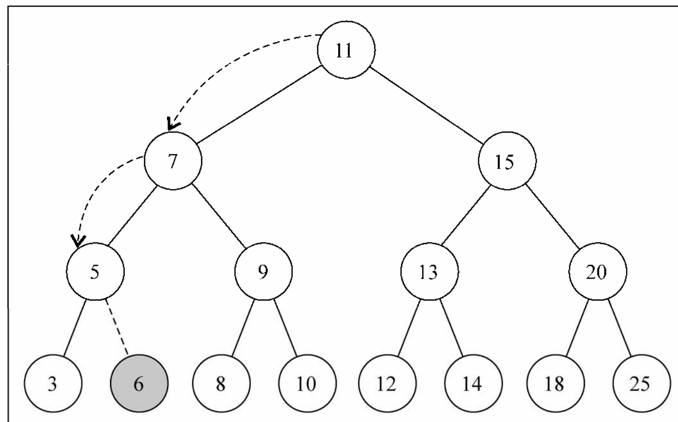
(2) 算法将会检测行{4}（key[6] < root[11]为真），并继续检测行{5}（node.left[7]不是null），然后将到达行{7}并调用insertNode（node.left[7], key[6]）。

(3) 将再次进入insertNode方法内部，但是使用了不同的参数。它会再次检测行{4}（key[6] < node[7]为真），然后再检测行{5}（node.left[5]不是null），接着到达行{7}，调用insertNode（node.left[5], key[6]）。

(4) 将再一次进入insertNode方法内部。它会再次检测行{4}（key[6] < node[5]为假），然后到达行{8}（node.right是null——节点5没有任何右侧的子节点），然后将会执行行{9}，在节点5的右侧子节点位置插入键6。

(5) 然后，方法调用会依次出栈，代码执行过程结束。

这是插入键6后的结果：



8.3　树的遍历

遍历一棵树是指访问树的每个节点并对它们进行某种操作的过程。但是我们应该怎么去做呢？应该从树的顶端还是底端开始呢？从左开始还是从右开始呢？访问树的所有节点有三种方式：中序、先序和后序。

在后面的小节中，我们将会深入了解这三种遍历方式的用法和实现。

8.3.1　中序遍历

中序遍历是一种以上行顺序访问BST所有节点的遍历方式，也就是以从最小到最大的顺序访问所有节点。中序遍历的一种应用就是对树进行排序操作。我们来看它的实现：

```
this.inOrderTraverse = function(callback){
    inOrderTraverseNode(root, callback); //{1}
};
```

inOrderTraverse方法接收一个回调函数作为参数。回调函数用来定义我们对遍历到的每个节点进行的操作（这也叫作访问者模式，要了解更多关于访问者模式的信息，请参考http://en.wikipedia.org/wiki/Visitor_pattern）。由于我们在BST中最常实现的算法是递归，这里使用了一个私有的辅助函数，来接收一个节点和对应的回调函数作为参数（行{1}）。

```
var inOrderTraverseNode = function (node, callback) {
    if (node !== null) { //{2}
        inOrderTraverseNode(node.left, callback);  //{3}
        callback(node.key);                        //{4}
        inOrderTraverseNode(node.right, callback); //{5}
    }
};
```

要通过中序遍历的方法遍历一棵树，首先要检查以参数形式传入的节点是否为null（这就是停止递归继续执行的判断条件——行{2}——递归算法的基本条件）。

然后，递归调用相同的函数来访问左侧子节点（行{3}）。接着对这个节点进行一些操作（callback），然后再访问右侧子节点（行{5}）。

我们试着在之前展示的树上执行下面的方法：

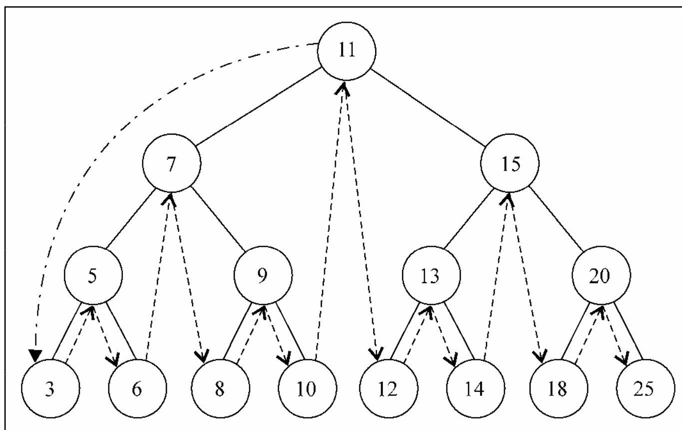
```
function printNode(value){ //{6}
    console.log(value);
}
tree.inOrderTraverse(printNode); //{7}
```

但首先，需要创建一个回调函数（行{6}）。我们要做的，是在浏览器的控制台上输出节点的值。然后，调用inOrderTraverse方法并将回调函数作为参数传入（行{7}）。当执行上面的

代码后，下面的结果将会在控制台上输出（每个数字将会输出在不同的行）：

```
3 5 6 7 8 9 10 11 12 13 14 15 18 20 25
```

下面的图描绘了inOrderTraverse方法的访问路径：



8.3.2 先序遍历

先序遍历是以优先于后代节点的顺序访问每个节点的。先序遍历的一种应用是打印一个结构化的文档。

我们来看实现：

```
this.preOrderTraverse = function(callback){
    preOrderTraverseNode(root, callback);
};
```

preOrderTraverseNode方法的实现如下：

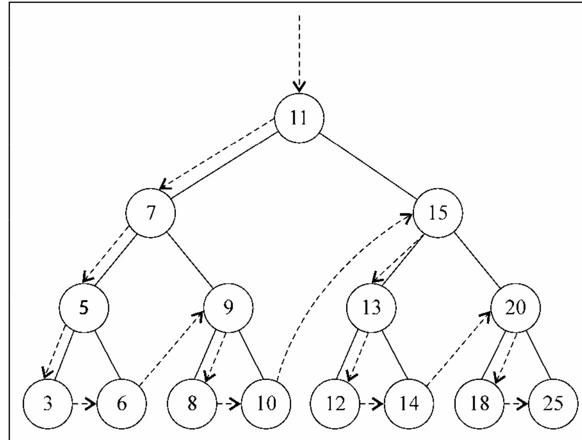
```
var preOrderTraverseNode = function (node, callback) {
    if (node !== null) {
        callback(node.key); //{1}
        preOrderTraverseNode(node.left, callback); //{2}
        preOrderTraverseNode(node.right, callback); //{3}
    }
};
```

先序遍历和中序遍历的不同点是，先序遍历会先访问节点本身（行{1}），然后再访问它的左侧子节点（行{2}），最后是右侧子节点（行{3}），而中序遍历的执行顺序是：{2}、{1}和{3}。

下面是控制台上的输出结果（每个数字将会输出在不同的行）：

```
11 7 5 3 6 9 8 10 15 13 12 14 20 18 25
```

下面的图描绘了preOrderTraverse方法的访问路径：



8.3.3　后序遍历

后序遍历则是先访问节点的后代节点，再访问节点本身。后序遍历的一种应用是计算一个目录和它的子目录中所有文件所占空间的大小。

我们来看它的实现：

```
this.postOrderTraverse = function(callback){
    postOrderTraverseNode(root, callback);
};
```

postOrderTraverseNode方法的实现如下：

```
var postOrderTraverseNode = function (node, callback) {
    if (node !== null) {
        postOrderTraverseNode(node.left, callback);  //{1}
        postOrderTraverseNode(node.right, callback); //{2}
        callback(node.key);                          //{3}
    }
};
```

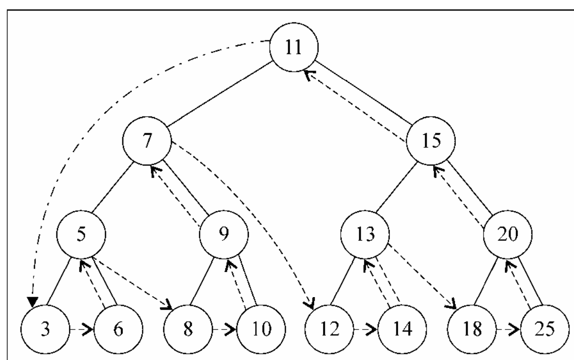
这个例子中，后序遍历会先访问左侧子节点（行{1}），然后是右侧子节点（行{2}），最后是父节点本身（行{3}）。

你会发现，中序、先序和后序遍历的实现方式是很相似的，唯一不同的是行{1}、{2}和{3}的执行顺序。

下面是控制台的输出结果（每个数字将会输出在不同行）：

```
3 6 5 8 10 9 7 12 14 13 18 25 20 15 11
```

下面的图描绘了`postOrderTraverse`方法的访问路径：



8.4　搜索树中的值

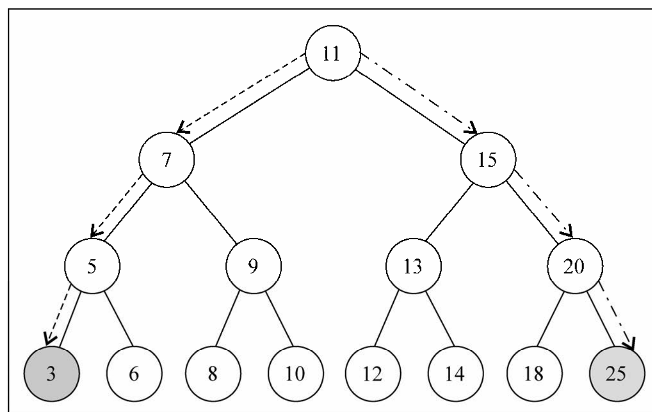
在树中，有三种经常执行的搜索类型：

- 最小值；
- 最大值；
- 搜索特定的值。

我们依次来看。

8.4.1　搜索最小值和最大值

我们使用下面的树作为示例：



只用眼睛看这张图，你能一下找到树中的最小值和最大值吗？

如果你看一眼树最后一层最左侧的节点，会发现它的值为3，这是这棵树中最小的键。如果你再看一眼树最右端的节点（同样是树的最后一层），会发现它的值为25，这是这棵树中最大的键。这条信息在我们实现搜索树节点的最小值和最大值的方法时能给予我们很大的帮助。

首先，我们来看寻找树的最小键的方法：

```
this.min = function() {
    return minNode(root); //{1}
};
```

min方法将会暴露给用户。这个方法调用了minNode方法（行{1}）：

```
var minNode = function (node) {
    if (node){
        while (node && node.left !== null) { //{2}
            node = node.left;                //{3}
        }

        return node.key;
    }
    return null;  //{4}
};
```

minNode方法允许我们从树中任意一个节点开始寻找最小的键。我们可以使用它来找到一棵树或它的子树中最小的键。因此，我们在调用minNode方法的时候传入树的根节点（行{1}），因为我们想要找到整棵树的最小键。

在minNode内部，我们会遍历树的左边（行{2}和行{3}）直到找到树的最下层（最左端）。

以相似的方式，可以实现max方法：

```
this.max = function() {
    return maxNode(root);
};

var maxNode = function (node) {
    if (node){
        while (node && node.right !== null) { //{5}
            node = node.right;
        }

        return node.key;
    }
    return null;
};
```

要找到最大的键，我们要沿着树的右边进行遍历（行{5}）直到找到最右端的节点。

因此，对于寻找最小值，总是沿着树的左边；而对于寻找最大值，总是沿着树的右边。

8.4.2　搜索一个特定的值

在之前的章节中，我们同样实现了find、search或get方法来查找数据结构中的一个特定的值（和之前章节中实现的has方法相似）。我们将同样在BST中实现搜索的方法，来看它的实现：

```
this.search = function(key){
    return searchNode(root, key); //{1}
};

var searchNode = function(node, key){

    if (node === null){ //{2}
        return false;
    }
    if (key < node.key){ //{3}
        return searchNode(node.left, key);  //{4}

    } else if (key > node.key){             //{5}

        return searchNode(node.right, key); //{6}
    } else {
        return true; //{7}
    }
};
```

我们要做的第一件事，是声明search方法。和BST中声明的其他方法的模式相同，我们将会使用一个辅助函数（行{1}）。

searchNode方法可以用来寻找一棵树或它的任意子树中的一个特定的值。这也是为什么在行{1}中调用它的时候传入树的根节点作为参数。

在开始算法之前，先要验证作为参数传入的node是否合法（不是null）。如果是null的话，说明要找的键没有找到，返回false。

如果传入的节点不是null，需要继续验证。如果要找的键比当前的节点小（行{3}），那么继续在左侧的子树上搜索（行{4}）。如果要找的键比当前的节点大，那么就从右侧子节点开始继续搜索（行{6}），否则就说明要找的键和当前节点的键相等，就返回true来表示找到了这个键（行{7}）。

可以通过下面的代码来测试这个方法：

```
console.log(tree.search(1) ? 'Key 1 found.' : 'Key 1 not found.');
console.log(tree.search(8) ? 'Key 8 found.' : 'Key 8 not found.');
```

输出结果如下：

```
Value 1 not found.
Value 8 found.
```

让我们详细展示查找1这个键的时候方法是如何执行的。

(1) 调用searchNode方法，传入根节点作为参数（行{1}）。（node[root[11]]）不是null（行{2}），因此我们执行到行{3}。

(2) （key[1] < node[11]）为ture（行{3}），因此来到行{4}并再次调用searchNode方法，传入（node[7], key[1]）作为参数。

(3) （node[7]）不是null（{2}），因此继续执行行{3}。

(4) （key[1] < node[7]）为ture（行{3}），因此来到行{4}并再次调用searchNode方法，传入（node[5], key[1]）作为参数。

(5) （node[5]）不是null（行{2}），因此继续执行行{3}。

(6) （key[1] < node[5]）为ture（行{3}），因此来到行{4}并再次调用searchNode方法，传入（node[3], key[1]）作为参数。

(7) （node[3]）不是null（行{2}），因此来到行{3}。

(8) （key[1] < node[3]）为真（行{3}），因此来到行{4}并再次调用searchNode方法，传入（null，key[1]）作为参数。null被作为参数传入是因为node[3]是一个叶节点（它没有子节点，所以它的左侧子节点的值为null）。

(9) 节点（null）的值为null（行{2}，这时要搜索的节点为null），因此返回false。

(10) 然后，方法调用会依次出栈，代码执行过程结束。

让我们再来查找值为8的节点：

(1) 调用searchNode方法，传入root作为参数（行{1}）。（node[root[11]]）不是null（行{2}），因此我们来到行{3}。

(2) （key[8] < node[11]）为真（行{3}），因此执行到行{4}并再次调用searchNode方法，传入（node[7], key[8]）作为参数。

(3) （node[7]）不是null，因此来到行{3}。

(4) （key[8] < node[7]）为假（行{3}），因此来到行{5}。

(5) （key[8] > node[7]）为真（行{5}），因此来到行{6}并再次调用searchNode方法，传入（node[9], key[8]）作为参数。

(6) （node[9]）不是null（行{2}），因此来到行{3}。

(7) （key[8] < node[9]）为真（行{3}），因此来到行{4}并再次调用searchNode方法，传入（node[8], key[8]）作为参数。

(8) （node[8]）不是null（行{2}），因此来到行{3}。

(9) （key[8] < node[8]）为假（行{3}），因此来到行{5}。

(10) （key[8] > node[8]）为假（行{5}），因此来到行{7}并返回true，因为node[8]就是要找的键。

(11) 然后，方法调用会依次出栈，代码执行过程结束。

8.4.3　移除一个节点

我们要为BST实现的下一个、也是最后一个方法是remove方法。这是我们在本书中要实现

的最复杂的方法。我们先创建这个方法，使它能够在树的实例上被调用：

```
this.remove = function(key){
    root = removeNode(root, key); //{1}
};
```

这个方法接收要移除的键并且它调用了removeNode方法，传入root和要移除的键作为参数（行{1}）。我要提醒大家的一件非常重要的事情是，root被赋值为removeNode方法的返回值。我们稍后会明白其中的原因。

removeNode方法的复杂之处在于我们要处理不同的运行场景，当然也包括它同样是通过递归来实现的。

我们来看removeNode方法的实现：

```
var removeNode = function(node, key){

    if (node === null){ //{2}
        return null;
    }
    if (key < node.key){ //{3}
        node.left = removeNode(node.left, key); //{4}
        return node; //{5}
    } else if (key > node.key){ //{6}
        node.right = removeNode(node.right, key); //{7}
        return node; //{8}

    } else { //键等于node.key

        //第一种情况——一个叶节点
        if (node.left === null && node.right === null){ //{9}
            node = null; //{10}
            return node; //{11}
        }

        //第二种情况——一个只有一个子节点的节点
        if (node.left === null){ //{12}
            node = node.right; //{13}
            return node; //{14}

        } else if (node.right === null){ //{15}
            node = node.left; //{16}
            return node; //{17}
        }

        //第三种情况——一个有两个子节点的节点
        var aux = findMinNode(node.right); //{18}
        node.key = aux.key; //{19}
        node.right = removeNode(node.right, aux.key); //{20}
        return node; //{21}
    }
};
```

我们来看行{2}，如果正在检测的节点是null，那么说明键不存在于树中，所以返回null。

然后，我们要做的第一件事，就是在树中找到要移除的节点。因此，如果要找的键比当前节点的值小（行{3}），就沿着树的左边找到下一个节点（行{4}）。如果要找的键比当前节点的值大（行{6}），那么就沿着树的右边找到下一个节点（行{7}）。

如果我们找到了要找的键（键和node.key相等），就需要处理三种不同的情况。

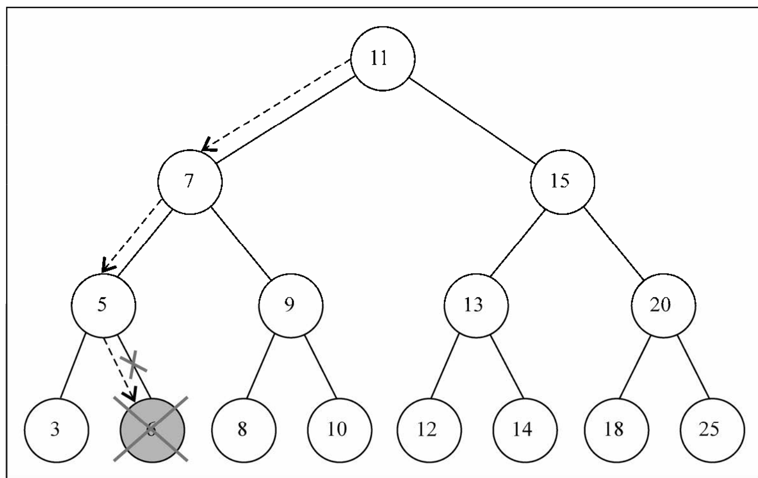
1. 移除一个叶节点

第一种情况是该节点是一个没有左侧或右侧子节点的叶节点——行{9}。在这种情况下，我们要做的就是给这个节点赋予null值来移除它（行{9}）。但是当学习了链表的实现之后，我们知道仅仅赋一个null值是不够的，还需要处理指针。在这里，这个节点没有任何子节点，但是它有一个父节点，需要通过返回null来将对应的父节点指针赋予null值（行{11}）。

现在节点的值已经是null了，父节点指向它的指针也会接收到这个值，这也是我们要在函数中返回节点的值的原因。父节点总是会接收到函数的返回值。另一种可行的办法是将父节点和节点本身都作为参数传入方法内部。

如果回头来看方法的第一行代码，会发现我们在行{4}和行{7}更新了节点左右指针的值，同样也在行{5}和行{8}返回了更新后的节点。

下图展现了移除一个叶节点的过程：



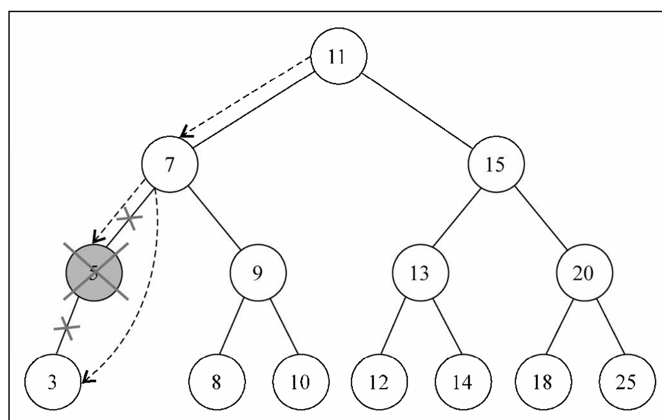
2. 移除有一个左侧或右侧子节点的节点

现在我们来看第二种情况，移除有一个左侧子节点或右侧子节点的节点。这种情况下，需要

跳过这个节点，直接将父节点指向它的指针指向子节点。

如果这个节点没有左侧子节点（行`{12}`），也就是说它有一个右侧子节点。因此我们把对它的引用改为对它右侧子节点的引用（行`{13}`）并返回更新后的节点（行`{14}`）。如果这个节点没有右侧子节点，也是一样——把对它的引用改为对它左侧子节点的引用（行`{16}`）并返回更新后的值（行`{17}`）。

下图展现了移除只有一个左侧子节点或右侧子节点的节点的过程：



3. 移除有两个子节点的节点

现在是第三种情况，也是最复杂的情况，那就是要移除的节点有两个子节点——左侧子节点和右侧子节点。要移除有两个子节点的节点，需要执行四个步骤。

(1) 当找到了需要移除的节点后，需要找到它右边子树中最小的节点（它的继承者——行`{18}`）。

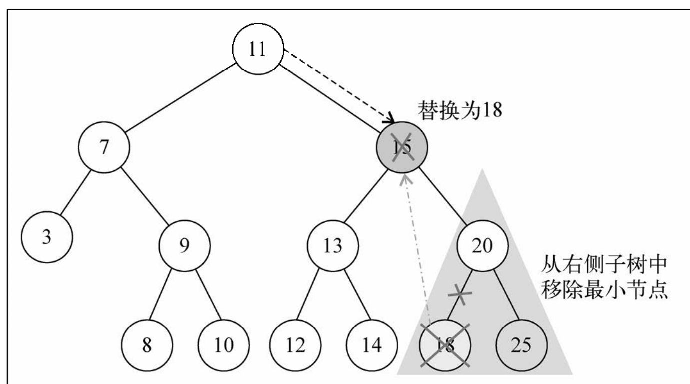
(2) 然后，用它右侧子树中最小节点的键去更新这个节点的值（行`{19}`）。通过这一步，我们改变了这个节点的键，也就是说它被移除了。

(3) 但是，这样在树中就有两个拥有相同键的节点了，这是不行的。要继续把右侧子树中的最小节点移除，毕竟它已经被移至要移除的节点的位置了（行`{20}`）。

(4) 最后，向它的父节点返回更新后节点的引用（行`{21}`）。

`findMinNode`方法的实现和`min`方法的实现方式是一样的。唯一不同之处在于，在`min`方法中只返回键，而在`findMinNode`中返回了节点。

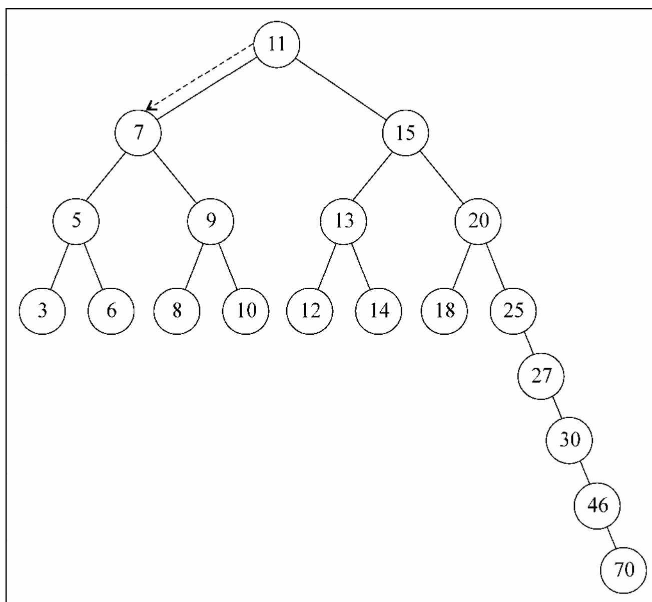
下图展现了移除有两个子节点的节点的过程：



8.5　更多关于二叉树的知识

现在你知道如何使用二叉搜索树了，如果愿意的话，可以继续去学习更多关于树的知识。

BST存在一个问题：取决于你添加的节点数，树的一条边可能会非常深；也就是说，树的一条分支会有很多层，而其他的分支却只有几层，如下图所示：



这会在需要在某条边上添加、移除和搜索某个节点时引起一些性能问题。为了解决这个问题，有一种树叫作阿德尔森-维尔斯和兰迪斯树（AVL树）。AVL树是一种自平衡二叉搜索树，意思是任何一个节点左右两侧子树的高度之差最多为1。也就是说这种树会在添加或移除节点时尽量试

着成为一棵完全树。

我们不会在本书中介绍AVL树的实现，但是你可以在本书源代码的chapter08文件夹中找到它的源代码。



另一种你同样应该学习的树是红黑树，它是一种特殊的二叉树。这种树可以进行高效的中序遍历（http://goo.gl/OxED8K）。此外，堆积树也值得你去学习（http://goo.gl/SFlhW6）。

8.6 小结

在本章中，我们介绍了在计算机科学中被广泛使用的基本树数据结构——二叉搜索树中添加、搜索和移除项的算法。我们同样介绍了访问树中每个节点的三种遍历方式。

在下一章中，我们将会学习图的基本概念，它也是一种非线性的数据结构。

第 9 章

图

在本章，你将学习另一种非线性数据结构——图。这是我们要讲的最后一种数据结构，下一章将深入学习排序和搜索算法。

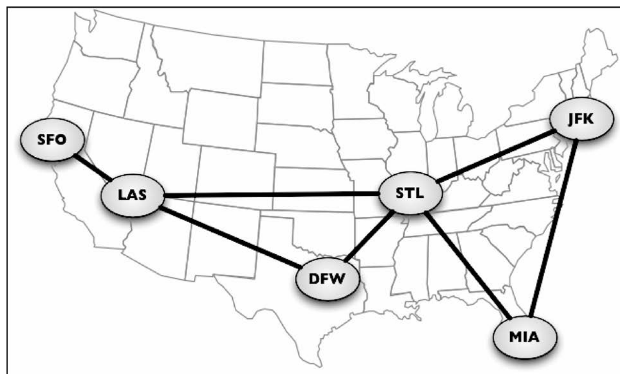
本章将会包含不少图的巧妙运用。图是一个庞大的主题，深入探索图的奇妙世界都足够写一本书了。

9.1 图的相关术语

图是网络结构的抽象模型。图是一组由边连接的节点（或顶点）。学习图是重要的，因为任何二元关系都可以用图来表示。

任何社交网络，例如Facebook、Twitter和Google plus，都可以用图来表示。

我们还可以使用图来表示道路、航班以及通信状态，如下图所示：

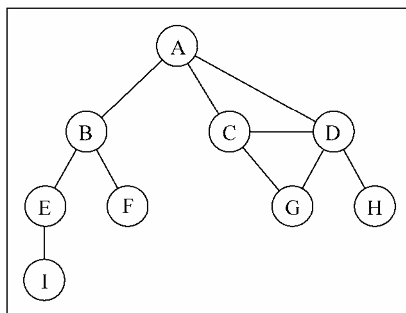


让我们来学习一下图在数学及技术上的概念。

一个图$G = (V, E)$由以下元素组成。

- V：一组顶点
- E：一组边，连接V中的顶点

下图表示一个图：



在着手实现算法之前，让我们先了解一下图的一些术语。

由一条边连接在一起的顶点称为*相邻顶点*。比如，A和B是相邻的，A和D是相邻的，A和C是相邻的，A和E不是相邻的。

一个顶点的*度*是其相邻顶点的数量。比如，A和其他三个顶点相连接，因此，A的度为3；E和其他两个顶点相连，因此，E的度为2。

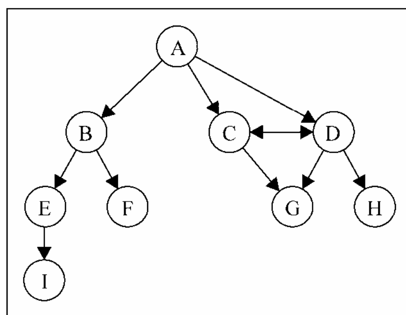
*路径*是顶点$v_1, v_2, \cdots, v_k$的一个连续序列，其中v_i和v_{i+1}是相邻的。以上一示意图中的图为例，其中包含路径A B E I和A C D G。

*简单路径*要求不包含重复的顶点。举个例子，A D G是一条简单路径。除去最后一个顶点（因为它和第一个顶点是同一个顶点），*环*也是一个简单路径，比如A D C A（最后一个顶点重新回到A）。

如果图中不存在环，则称该图是*无环的*。如果图中每两个顶点间都存在路径，则该图是*连通的*。

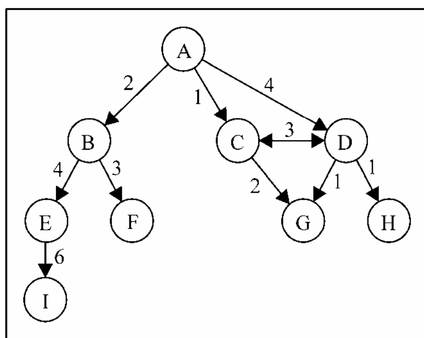
有向图和无向图

图可以是*无向的*（边没有方向）或是*有向的*（有向图）。如下图所示，有向图的边有一个方向：



如果图中每两个顶点间在双向上都存在路径，则该图是强连通的。例如，C和D是强连通的，而A和B不是强连通的。

图还可以是未加权的（目前为止我们看到的图都是未加权的）或是加权的。如下图所示，加权图的边被赋予了权值：



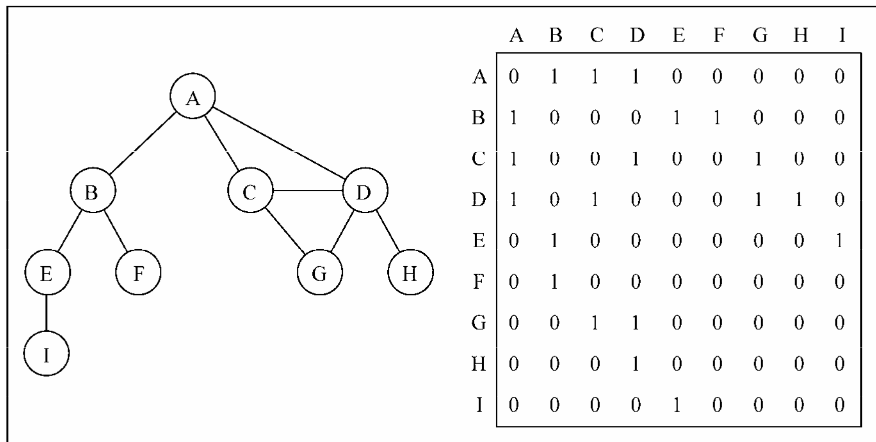
我们可以使用图来解决计算机科学世界中的很多问题，比如搜索图中的一个特定顶点或搜索一条特定边，寻找图中的一条路径（从一个顶点到另一个顶点），寻找两个顶点之间的最短路径，以及环检测。

9.2　图的表示

从数据结构的角度来说，我们有多种方式来表示图。在所有的表示法中，不存在绝对正确的方式。图的正确表示法取决于待解决的问题和图的类型。

9.2.1　邻接矩阵

图最常见的实现是邻接矩阵。每个节点都和一个整数相关联，该整数将作为数组的索引。我们用一个二维数组来表示顶点之间的连接。如果索引为*i*的节点和索引为*j*的节点相邻，则array[*i*][*j*] === 1，否则array[*i*][*j*] === 0，如下图所示：

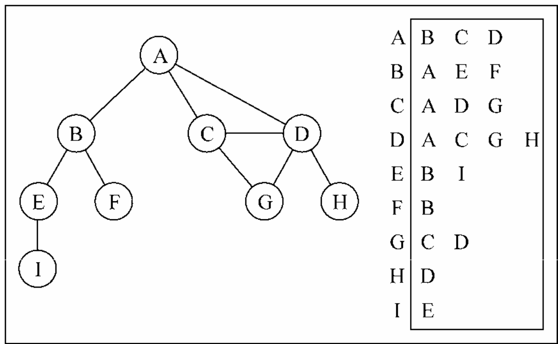


	A	B	C	D	E	F	G	H	I
A	0	1	1	1	0	0	0	0	0
B	1	0	0	0	1	1	0	0	0
C	1	0	0	1	0	0	1	0	0
D	1	0	1	0	0	0	1	1	0
E	0	1	0	0	0	0	0	0	1
F	0	1	0	0	0	0	0	0	0
G	0	0	1	1	0	0	0	0	0
H	0	0	0	1	0	0	0	0	0
I	0	0	0	0	1	0	0	0	0

不是强连通的图（稀疏图）如果用邻接矩阵来表示，则矩阵中将会有很多0，这意味着我们浪费了计算机存储空间来表示根本不存在的边。例如，找给定顶点的相邻顶点，即使该顶点只有一个相邻顶点，我们也不得不迭代一整行。邻接矩阵表示法不够好的另一个理由是，图中顶点的数量可能会改变，而2维数组不太灵活。

9.2.2 邻接表

我们也可以使用一种叫作邻接表的动态数据结构来表示图。邻接表由图中每个顶点的相邻顶点列表所组成。存在好几种方式来表示这种数据结构。我们可以用列表（数组）、链表，甚至是散列表或是字典来表示相邻顶点列表。下面的示意图展示了邻接表数据结构。

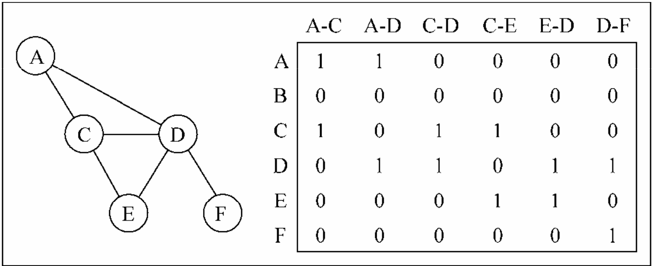


A	B	C	D	
B	A	E	F	
C	A	D	G	
D	A	C	G	H
E	B	I		
F	B			
G	C	D		
H	D			
I	E			

尽管邻接表可能对大多数问题来说都是更好的选择，但以上两种表示法都很有用，且它们有着不同的性质（例如，要找出顶点v和w是否相邻，使用邻接矩阵会比较快）。在本书的示例中，我们将会使用邻接表表示法。

9.2.3　关联矩阵

我们还可以用关联矩阵来表示图。在关联矩阵中，矩阵的行表示顶点，列表示边。如下图所示，我们使用二维数组来表示两者之间的连通性，如果顶点*v*是边*e*的入射点，则array[*v*][*e*] === 1；否则，array[*v*][*e*] === 0。



	A-C	A-D	C-D	C-E	E-D	D-F
A	1	1	0	0	0	0
B	0	0	0	0	0	0
C	1	0	1	1	0	0
D	0	1	1	0	1	1
E	0	0	0	1	1	0
F	0	0	0	0	0	1

关联矩阵通常用于边的数量比顶点多的情况下，以节省空间和内存。

9.3　创建图类

照例，我们声明类的骨架：

```
function Graph() {
    var vertices = []; //{1}
    var adjList = new Dictionary(); //{2}
}
```

我们使用一个数组来存储图中所有顶点的名字（行{1}），以及一个字典（在第7章中已经实现）来存储邻接表（行{2}）。字典将会使用顶点的名字作为键，邻接顶点列表作为值。vertices数组和adjList字典两者都是我们Graph类的私有属性。

接着，我们将实现两个方法：一个用来向图中添加一个新的顶点（因为图实例化后是空的），另外一个方法用来添加顶点之间的边。我们先实现addVertex方法：

```
this.addVertex = function(v){
   vertices.push(v); //{3}
   adjList.set(v, []); //{4}
};
```

这个方法接受顶点v作为参数。我们将该顶点添加到顶点列表中（行{3}），并且在邻接表中，设置顶点v作为键对应的字典值为一个空数组（行{4}）。

现在，我们来实现addEdge方法：

```
this.addEdge = function(v, w){
   adjList.get(v).push(w); //{5}
```

```
    adjList.get(w).push(v); //{6}
};
```

这个方法接受两个顶点作为参数。首先，通过将w加入到v的邻接表中，我们添加了一条自顶点v到顶点w的边。如果你想实现一个有向图，则行{5}就足够了。由于本章中大多数的例子都是基于无向图的，我们需要添加一条自w向v的边（行{6}）。

请注意我们只是往数组里新增元素，因为数组已经在行{4}被初始化了。

让我们测试这段代码：

```
var graph = new Graph();
var myVertices = ['A','B','C','D','E','F','G','H','I']; //{7}
for (var i=0; i<myVertices.length; i++){ //{8}
    graph.addVertex(myVertices[i]);
}
graph.addEdge('A', 'B'); //{9}
graph.addEdge('A', 'C');
graph.addEdge('A', 'D');
graph.addEdge('C', 'D');
graph.addEdge('C', 'G');
graph.addEdge('D', 'G');
graph.addEdge('D', 'H');
graph.addEdge('B', 'E');
graph.addEdge('B', 'F');
graph.addEdge('E', 'I');
```

为方便起见，我们创建了一个数组，包含所有我们想添加到图中的顶点（行{7}）。接下来，我们只要遍历vertices数组并将其中的值逐一添加到我们的图中（行{8}）。最后，我们添加想要的边（行{9}）。这段代码将会创建一个图，也就是到目前为止本章的示意图所使用的。

为了更方便一些，让我们来实现一下Graph类的toString方法，以便于在控制台输出图。

```
this.toString = function(){
    var s = '';
    for (var i=0; i<vertices.length; i++){ //{10}
        s += vertices[i] + ' -> ';
        var neighbors = adjList.get(vertices[i]); //{11}
        for (var j=0; j<neighbors.length; j++){ //{12}
            s += neighbors[j] + ' ';
        }
        s += '\n'; //{13}
    }
    return s;
};
```

我们为邻接表表示法构建了一个字符串。首先，迭代vertices数组列表（行{10}），将顶点的名字加入字符串中。接着，取得该顶点的邻接表（行{11}），同样也迭代该邻接表（行{12}），将相邻顶点加入我们的字符串。邻接表迭代完成后，给我们的字符串添加一个换行符（行{13}），这样就可以在控制台看到一个漂亮的输出了。运行如下代码：

```
console.log(graph.toString());
```

输出如下：

```
A -> B C D
B -> A E F
C -> A D G
D -> A C G H
E -> B I
F -> B
G -> C D
H -> D
I -> E
```

一个漂亮的邻接表！从该输出中，我们知道顶点A有这几个相邻顶点：B、C和D。

9.4　图的遍历

和树数据结构类似，我们可以访问图的所有节点。有两种算法可以对图进行遍历：广度优先搜索（Breadth-First Search，BFS）和深度优先搜索（Depth-First Search，DFS）。图遍历可以用来寻找特定的顶点或寻找两个顶点之间的路径，检查图是否连通，检查图是否含有环等。

在实现算法之前，让我们来更好地理解一下图遍历的思想方法。

图遍历算法的思想是必须追踪每个第一次访问的节点，并且追踪有哪些节点还没有被完全探索。对于两种图遍历算法，都需要明确指出第一个被访问的顶点。

完全探索一个顶点要求我们查看该顶点的每一条边。对于每一条边所连接的没有被访问过的顶点，将其标注为被发现的，并将其加进待访问顶点列表中。

为了保证算法的效率，务必访问每个顶点至多两次。连通图中每条边和顶点都会被访问到。

广度优先搜索算法和深度优先搜索算法基本上是相同的，只有一点不同，那就是待访问顶点列表的数据结构。

算　法	数据结构	描　述
深度优先搜索	栈	通过将顶点存入栈中（在第3章中学习过），顶点是沿着路径被探索的，存在新的相邻顶点就去访问
广度优先搜索	队列	通过将顶点存入队列中（在第4章中学习过），最先入队列的顶点先被探索

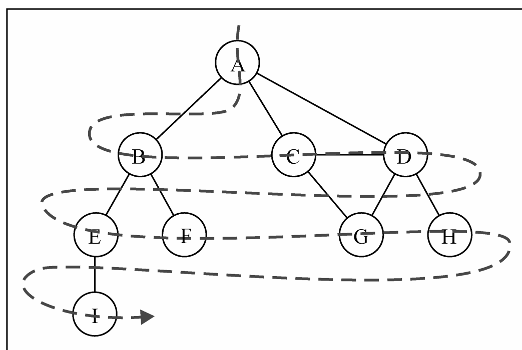
当要标注已经访问过的顶点时，我们用三种颜色来反映它们的状态。

- **白色**：表示该顶点还没有被访问。
- **灰色**：表示该顶点被访问过，但并未被探索过。
- **黑色**：表示该顶点被访问过且被完全探索过。

这就是之前提到的务必访问每个顶点最多两次的原因。

9.4.1 广度优先搜索

广度优先搜索算法会从指定的第一个顶点开始遍历图，先访问其所有的相邻点，就像一次访问图的一层。换句话说，就是先宽后深地访问顶点，如下图所示：



以下是从顶点v开始的广度优先搜索算法所遵循的步骤。

(1) 创建一个队列Q。
(2) 将v标注为被发现的（灰色），并将v入队列Q。
(3) 如果Q非空，则运行以下步骤：
  (a) 将u从Q中出队列；
  (b) 将标注u为被发现的（灰色）；
  (c) 将u所有未被访问过的邻点（白色）入队列；
  (d) 将u标注为已被探索的（黑色）。

让我们来实现广度优先搜索算法：

```
var initializeColor = function(){
    var color = [];
    for (var i=0; i<vertices.length; i++){
         color[vertices[i]] = 'white'; //{1}
    }
    return color;
};

this.bfs = function(v, callback){

   var color = initializeColor(),  //{2}
       queue = new Queue();        //{3}
   queue.enqueue(v);               //{4}

   while (!queue.isEmpty()){       //{5}
        var u = queue.dequeue(),   //{6}
            neighbors = adjList.get(u); //{7}
```

9

```
        color[u] = 'grey';                        // {8}
        for (var i=0; i<neighbors.length; i++){ // {9}
            var w = neighbors[i];               // {10}
            if (color[w] === 'white'){          // {11}
                color[w] = 'grey';              // {12}
                queue.enqueue(w);               // {13}
            }
        }
        color[u] = 'black'; // {14}
        if (callback) {     // {15}
            callback(u);
        }
    }
};
```

广度优先搜索和深度优先搜索都需要标注被访问过的顶点。为此，我们将使用一个辅助数组`color`。由于当算法开始执行时，所有的顶点颜色都是白色（行`{1}`），所以我们可以创建一个辅助函数`initializeColor`，为这两个算法执行此初始化操作。

让我们深入学习广度优先搜索方法的实现。我们要做的第一件事情是用`initializeColor`函数来将`color`数组初始化为`white`（行`{2}`）。我们还需要声明和创建一个`Queue`实例（行`{3}`），它将会存储待访问和待探索的顶点。

照着本章开头解释过的步骤，`bfs`方法接受一个顶点作为算法的起始点。起始顶点是必要的，我们将此顶点入队列（行`{4}`）。

如果队列非空（行`{5}`），我们将通过出队列（行`{6}`）操作从队列中移除一个顶点，并取得一个包含其所有邻点的邻接表（行`{7}`）。该顶点将被标注为`grey`（行`{8}`），表示我们发现了它（但还未完成对其的探索）。

对于u（行`{9}`）的每个邻点，我们取得其值（该顶点的名字——行`{10}`），如果它还未被访问过（颜色为`white`——行`{11}`），则将其标注为我们已经发现了它（颜色设置为`grey`——行`{12}`），并将这个顶点加入队列中（行`{13}`），这样当其从队列中出列的时候，我们可以完成对其的探索。

当完成探索该顶点和其相邻顶点后，我们将该顶点标注为已探索过的（颜色设置为`black`——行`{14}`）。

我们实现的这个`bfs`方法也接受一个回调（我们在第8章中遍历树时使用了一个相似的方法）。这个参数是可选的，如果我们传递了回调函数（行`{15}`），会用到它。

让我们执行下面这段代码来测试一下这个算法：

```
function printNode(value){ //{16}
    console.log('Visited vertex: ' + value); //{17}
}
graph.bfs(myVertices[0], printNode); //{18}
```

首先，我们声明了一个回调函数（行`{16}`），它仅仅在浏览器控制台上输出已经被完全探索过的顶点的名字。接着，我们会调用`bfs`方法，给它传递第一个顶点（A——从本章开头声明的`myVertices`数组）和回调函数。当我们执行这段代码时，该算法会在浏览器控制台输出下示的结果：

```
Visited vertex: A
Visited vertex: B
Visited vertex: C
Visited vertex: D
Visited vertex: E
Visited vertex: F
Visited vertex: G
Visited vertex: H
Visited vertex: I
```

如你所见，顶点被访问的顺序和本节开头的示意图中所展示的一致。

1. 使用BFS寻找最短路径

到目前为止，我们只展示了BFS算法的工作原理。我们可以用该算法做更多事情，而不只是输出被访问顶点的顺序。例如，考虑如何来解决下面这个问题。

给定一个图G和源顶点v，找出对每个顶点u，u和v之间最短路径的距离（以边的数量计）。

对于给定顶点v，广度优先算法会访问所有与其距离为1的顶点，接着是距离为2的顶点，以此类推。所以，可以用广度优先算法来解这个问题。我们可以修改`bfs`方法以返回给我们一些信息：

- 从v到u的距离$d[u]$；
- 前溯点pred[u]，用来推导出从v到其他每个顶点u的最短路径。

让我们来看看改进过的广度优先方法的实现：

```
this.BFS = function(v){

    var color = initializeColor(),
        queue = new Queue(),
        d = [],    //{1}
       pred = []; //{2}
    queue.enqueue(v);

    for (var i=0; i<vertices.length; i++){ //{3}
        d[vertices[i]] = 0;                //{4}
        pred[vertices[i]] = null;          //{5}
```

9

```
    }

    while (!queue.isEmpty()){
        var u = queue.dequeue(),
            neighbors = adjList.get(u);
        color[u] = 'grey';
        for (i=0; i<neighbors.length; i++){
            var w = neighbors[i];
            if (color[w] === 'white'){
                color[w] = 'grey';
                d[w] = d[u] + 1;           //{6}
               pred[w] = u;                //{7}
                queue.enqueue(w);
            }
        }
        color[u] = 'black';
    }
    return { //{8}
        distances: d,
        predecessors: pred
    };
};
```

这个版本的BFS方法有些什么改变?

[本章源代码中包含两个bfs方法：bfs（第一个）和BFS（改进版）。]

我们还需要声明数组d（行{1}）来表示距离，以及pred数组来表示前溯点。下一步则是对图中的每一个顶点，用0来初始化数组d（行{4}），用null来初始化数组pred。

当我们发现顶点u的邻点w时，则设置w的前溯点值为u（行{7}）。我们还通过给d[u]加1来设置v和w之间的距离（u是w的前溯点，d[u]的值已经有了）。

方法最后返回了一个包含d和pred的对象（行{8}）。

现在，我们可以再次执行BFS方法，并将其返回值存在一个变量中：

```
var shortestPathA = graph.BFS(myVertices[0]);
console.log(shortestPathA);
```

对顶点A执行BFS方法，以下将会是输出：

```
distances: [A: 0, B: 1, C: 1, D: 1, E: 2, F: 2, G: 2, H: 2 , I: 3],
predecessors: [A: null, B: "A", C: "A", D: "A", E: "B", F: "B", G:"C", H: "D", I: "E"]
```

这意味着顶点A与顶点B、C和D的距离为1；与顶点E、F、G和H的距离为2；与顶点I的距离为3。

通过前溯点数组，我们可以用下面这段代码来构建从顶点A到其他顶点的路径：

```
var fromVertex = myVertices[0]; //{9}
for (var i=1; i<myVertices.length; i++){ //{10}
    var toVertex = myVertices[i], //{11}
        path = new Stack();       //{12}
    for (var v=toVertex; v!== fromVertex;
          v=shortestPathA.predecessors[v]) { //{13}
        path.push(v);                       //{14}
    }
    path.push(fromVertex);        //{15}
    var s = path.pop();           //{16}
    while (!path.isEmpty()){      //{17}
        s += ' - ' + path.pop();  //{18}
    }
    console.log(s); //{19}
}
```

我们用顶点A作为源顶点（行{9}）。对于每个其他顶点（除了顶点A——行{10}），我们会计算顶点A到它的路径。我们从顶点数组得到toVertex（行{11}），然后会创建一个栈来存储路径值（行{12}）。

接着，我们追溯toVertex到fromVertex的路径{行{13}}。变量v被赋值为其前溯点的值，这样我们能够反向追溯这条路径。将变量v添加到栈中（行{14}）。最后，源顶点也会被添加到栈中，以得到完整路径。

这之后，我们创建了一个s字符串，并将源顶点赋值给它（它是最后一个加入栈中的，所以它是第一个被弹出的项——行{16}）。当栈是非空的，我们就从栈中移出一个项并将其拼接到字符串s的后面（行{18}）。最后（行{19}）在控制台上输出路径。

执行该代码段，我们会得到如下输出：

```
A - B
A - C
A - D
A - B - E
A - B - F
A - C - G
A - D - H
A - B - E - I
```

这里，我们得到了从顶点A到图中其他顶点的最短路径（衡量标准是边的数量）。

2. 深入学习最短路径算法

本章中的图不是加权图。如果要计算加权图中的最短路径（例如，城市A和城市B之间的最短路径——GPS和Google Maps中用到的算法），广度优先搜索未必合适。

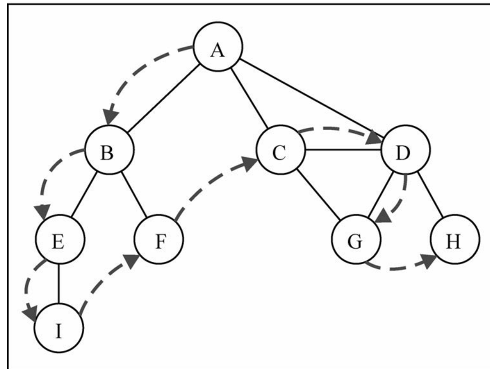
举些例子，Dijkstra's算法解决了单源最短路径问题。Bellman–Ford算法解决了边权值为负的单源最短路径问题。A*搜索算法解决了求仅一对顶点间的最短路径问题，它用经验法则来加速

搜索过程。Floyd–Warshall算法解决了求所有顶点对间的最短路径这一问题。

如本章开头提到的，图是一个广泛的主题，对最短路径问题及其变种问题，我们有很多的解决方案。但在开始学习这些其他解决方案前，我们需要掌握好图的基本概念，这是本章涵盖的内容。而这些其他解决方案则不会在本章讲述，但你可以自行探索图的奇妙世界。

9.4.2　深度优先搜索

深度优先搜索算法将会从第一个指定的顶点开始遍历图，沿着路径直到这条路径最后一个顶点被访问了，接着原路回退并探索下一条路径。换句话说，它是先深度后广度地访问顶点，如下图所示：



深度优先搜索算法不需要一个源顶点。在深度优先搜索算法中，若图中顶点v未访问，则访问该顶点v。

要访问顶点v，照如下步骤做。

(1) 标注v为被发现的（灰色）。

(2) 对于v的所有未访问的邻点w：

　　(a) 访问顶点w。

(3) 标注v为已被探索的（黑色）。

如你所见，深度优先搜索的步骤是递归的，这意味着深度优先搜索算法使用栈来存储函数调用（由递归调用所创建的栈）。

让我们来实现一下深度优先算法：

```
this.dfs = function(callback){
    var color = initializeColor(); //{1}
```

```
    for (var i=0; i<vertices.length; i++){ //{2}
        if (color[vertices[i]] === 'white'){ //{3}
            dfsVisit(vertices[i], color, callback); //{4}
        }
    }
};

var dfsVisit = function(u, color, callback){
    color[u] = 'grey'; //{5}
    if (callback) {    //{6}
        callback(u);
    }
    var neighbors = adjList.get(u);          //{7}
    for (var i=0; i<neighbors.length; i++){ //{8}
        var w = neighbors[i];                //{9}
        if (color[w] === 'white'){           //{10}
            dfsVisit(w, color, callback);    //{11}
        }
    }
    color[u] = 'black'; //{12}
};
```

首先，我们创建颜色数组（行{1}），并用值white为图中的每个顶点对其做初始化，广度优先搜索也这么做的。接着，对于图实例中每一个未被访问过的顶点（行{2}和{3}），我们调用私有的递归函数dfsVisit，传递的参数为顶点、颜色数组以及回调函数（行{4}）。

当访问u顶点时，我们标注其为被发现的（grey——行{5}）。如果有callback函数的话（行{6}），则执行该函数输出已访问过的顶点。接下来一步是取得包含顶点u所有邻点的列表（行{7}）。对于顶点u的每一个未被访问过（颜色为white——行{10}和行{8}）的邻点w（行{9}），我们将调用dfsVisit函数，传递w和其他参数（行{11}——添加顶点w入栈，这样接下来就能访问它）。最后，在该顶点和邻点按深度访问之后，我们回退，意思是该顶点已被完全探索，并将其标注为black（行{12}）。

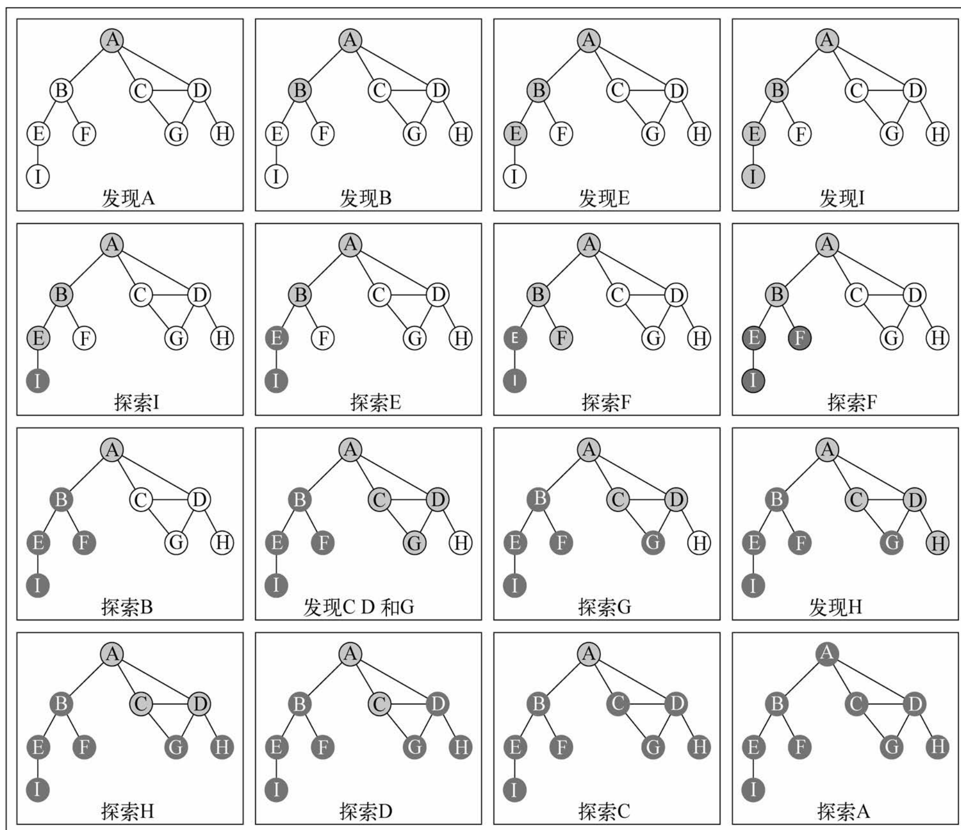
让我们执行下面的代码段来测试一下dfs方法：

```
graph.dfs(printNode);
```

输出如下：

```
Visited vertex: A
Visited vertex: B
Visited vertex: E
Visited vertex: I
Visited vertex: F
Visited vertex: C
Visited vertex: D
Visited vertex: G
Visited vertex: H
```

这个顺序和本节开头处示意图所展示的一致。下面这个示意图展示了该算法每一步的执行过程：



在我们示例所用的图中，行{4}只会被执行一次，因为所有其他的顶点都有路径到第一个调用dfsVisit函数的顶点（顶点A）。如果顶点B第一个调用函数，则行{4}将会为其他顶点再执行一次（比如顶点A）。

1. 探索深度优先算法

到目前为止，我们只是展示了深度优先搜索算法的工作原理。我们可以用该算法做更多的事情，而不只是输出被访问顶点的顺序。

对于给定的图G，我们希望深度优先搜索算法遍历图G的所有节点，构建"森林"（有根树的一个集合）以及一组源顶点（根），并输出两个数组：发现时间和完成探索时间。我们可以修改dfs方法来返回给我们一些信息：

- 顶点u的发现时间$d[u]$；
- 当顶点u被标注为黑色时，u的完成探索时间$f[u]$；

- ❑ 顶点u的前溯点$p[u]$。

让我们来看看改进了的DFS方法的实现：

```
var time = 0; //{1}
this.DFS = function(){
    var color = initializeColor(), //{2}
        d = [],
        f = [],
        p = [];
    time = 0;

    for (var i=0; i<vertices.length; i++){ //{3}
        f[vertices[i]] = 0;
        d[vertices[i]] = 0;
        p[vertices[i]] = null;
    }
    for (i=0; i<vertices.length; i++){
        if (color[vertices[i]] === 'white'){
            DFSVisit(vertices[i], color, d, f, p);
        }
    }
    return {           //{4}
        discovery: d,
        finished: f,
        predecessors: p
    };
};

var DFSVisit = function(u, color, d, f, p){
    console.log('discovered ' + u);
    color[u] = 'grey';
    d[u] = ++time; //{5}
    var neighbors = adjList.get(u);
    for (var i=0; i<neighbors.length; i++){
        var w = neighbors[i];
        if (color[w] === 'white'){
            p[w] = u;                         // {6}
            DFSVisit(w,color, d, f, p);
        }
    }
    color[u] = 'black';
    f[u] = ++time;      //{7}
    console.log('explored ' + u);
};
```

我们需要一个变量来要追踪发现时间和完成探索时间（行{1}）。时间变量不能被作为参数传递，因为非对象的变量不能作为引用传递给其他JavaScript方法（将变量作为引用传递的意思是如果该变量在其他方法内部被修改，新值会在原始变量中反映出来）。接下来，我们声明数组d、f和p（行{2}）。我们需要为图的每一个顶点来初始化这些数组（行{3}）。在这个方法结尾处返回这些值（行{4}），之后我们要用到它们。

当一个顶点第一次被发现时，我们追踪其发现时间（行{5}）。当它是由引自顶点u的边而被发现的，我们追踪它的前溯点（行{6}）。最后，当这个顶点被完全探索后，我们追踪其完成时间（行{7}）。

深度优先算法背后的思想是什么？边是从最近发现的顶点u处被向外探索的。只有连接到未发现的顶点的边被探索了。当u所有的边都被探索了，该算法回退到u被发现的地方去探索其他的边。这个过程持续到我们发现了所有从原始顶点能够触及的顶点。如果还留有任何其他未被发现的顶点，我们对新源顶点重复这个过程。重复该算法，直到图中所有的顶点都被探索了。

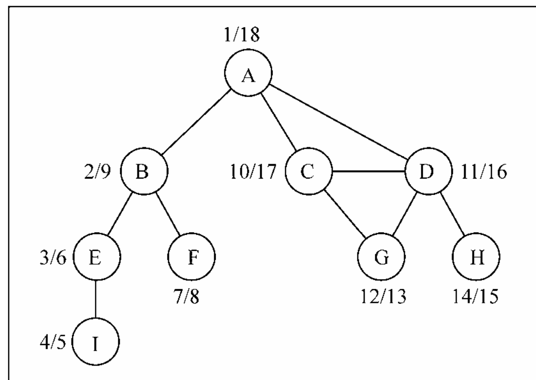
对于改进过的深度优先搜索，有两点需要我们注意：

- 时间（time）变量值的范围只可能在图顶点数量的一倍到两倍之间；
- 对于所有的顶点u，$d[u]<f[u]$（意味着，发现时间的值比完成时间的值小，完成时间意思是所有顶点都已经被探索过了）。

在这两个假设下，我们有如下的规则：

$$1 \leqslant d[u] < f[u] \leqslant 2|V|$$

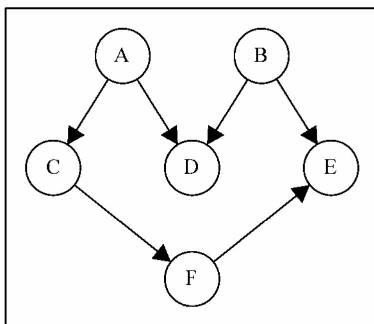
如果对同一个图再跑一遍新的深度优先搜索方法，对图中每个顶点，我们会得到如下的发现/完成时间：



但我们能用这些新信息来做什么呢？来看下一节。

2. 拓扑排序——使用深度优先搜索

给定下图，假定每个顶点都是一个我们需要去执行的任务：





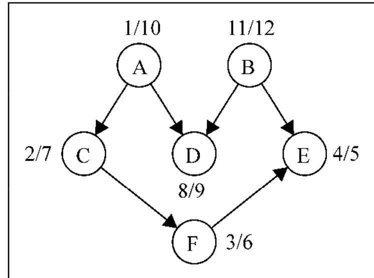
这是一个有向图，意味着任务的执行是有顺序的。例如，任务*F*不能在任务*A*之前执行。注意这个图没有环，意味着这是一个无环图。所以，我们可以说该图是一个有向无环图（DAG）。

当我们需要编排一些任务或步骤的执行顺序时，这称为拓扑排序（topological sorting，英文亦写作topsort或是toposort）。在日常生活中，这个问题在不同情形下都会出现。例如，当我们开始学习一门计算机科学课程，在学习某些知识之前得按顺序完成一些知识储备（你不可以在上算法I前先上算法II）。当我们在开发一个项目时，需要按顺序执行一些步骤，例如，首先我们得从客户那里得到需求，接着开发客户要求的东西，最后交付项目。你不能先交付项目再去收集需求。

拓扑排序只能应用于DAG。那么，如何使用深度优先搜索来实现拓扑排序呢？让我们在本节开头的示意图上执行一下深度优先搜索。

```
graph = new Graph();
myVertices = ['A','B','C','D','E','F'];
for (i=0; i<myVertices.length; i++){
    graph.addVertex(myVertices[i]);
}
graph.addEdge('A', 'C');
graph.addEdge('A', 'D');
graph.addEdge('B', 'D');
graph.addEdge('B', 'E');
graph.addEdge('C', 'F');
graph.addEdge('F', 'E');
var result = graph.DFS();
```

这段代码将创建图，添加边，执行改进版本的深度优先搜索算法，并将结果保存到`result`变量。下图展示了深度优先搜索算法执行后，该图的发现和完成时间。



现在要做的仅仅是以倒序来排序完成时间数组，这便得出了该图的拓扑排序：

```
B - A - D - C - F - E
```

注意之前的拓扑排序结果仅是多种可能性之一。如果我们稍微修改一下算法，就会有不同的结果，比如下面这个结果也是众多其他可能性中的一个：

```
A - B - C - D - F - E
```

这也是一个可以接受的结果。

9.5　小结

本章涵盖了图的基本概念。我们学习了几种不同的方式来表示这一数据结构，并实现了用邻接表表示图的算法。你还学到了如何用广度优先搜索和深度优先搜索来遍历图。本章还包括了广度优先搜索和深度优先搜索的两个实际应用，它们分别是使用广度优先搜索来找到最短路径，以及使用深度优先搜索来做拓扑排序。

下一章，我们将会学习计算机科学中最常用的排序算法。

第 10 章

排序和搜索算法

10

假设我们有一个没有任何排列顺序的电话号码表（或号码簿）。当需要添加联络人和电话时，你只能将其写在下一个空位上。假定你的联系人列表上有很多人，某天，你要找某个联系人及其电话号码。但是由于联系人列表没有按照任何顺序来组织，你只能逐个检查，直到找到那个你想要的联系人为止。这个方法太吓人了，难道你不这么认为？想象一下你要在黄页上搜寻一个联系人，但是那本黄页没有进行任何组织，那得花多久时间啊？！

因此（还有其他原因），我们需要组织信息集，比如那些存储在数据结构里的信息。排序和搜索算法广泛地运用在待解决的日常问题中。本章，你会学到最常用的排序和搜索算法。

10.1 排序算法

从上面的引言中，你应该理解，对给定信息得先排序再搜索。本节会介绍一些在计算机科学中最著名的排序算法。我们会从最慢的一个开始，接着是一些性能较好的算法。

在开始排序算法之前，我们先创建一个数组（列表）来表示待排序和搜索的数据结构。

```
function ArrayList(){

    var array = []; //{1}

    this.insert = function(item){ //{2}
        array.push(item);
    };

    this.toString= function(){ //{3}
        return array.join();
    };
}
```


如你所见，`ArrayList`是一个简单的数据结构，它将项存储在数组中（行`{1}`）。我们只需要一个插入方法来向数据结构中添加元素（行`{2}`），用第2章中介绍的JavaScript `Array`类原生的`push`方法即可。最后，为了帮助我们验证结果，`toString`方法使用JavaScript原生`Array`类的`join`方法，来拼接数组中的所有元素至一个单一的字符串，这样我们就可以轻松地在浏览器的控制台

输出结果了。



> join方法拼接数组元素至一个字符串，并返回该字符串。

注意ArrayList类并没有任何方法来移除数据或插入数据到特定位置。我们刻意保持简单是为了能够专注于排序和搜索算法。所有的排序和搜索算法会添加至这个类中。

我们现在开始吧！

10.1.1　冒泡排序

人们开始学习排序算法时，通常都先学冒泡算法，因为它在所有排序算法中最简单。然而，从运行时间的角度来看，冒泡排序是最差的一个，接下来你会知晓原因。

冒泡排序比较任何两个相邻的项，如果第一个比第二个大，则交换它们。元素项向上移动至正确的顺序，就好像气泡升至表面一样，冒泡排序因此得名。

让我们来实现一下冒泡排序：

```
this.bubbleSort = function(){
    var length = array.length;            //{1}
    for (var i=0; i<length; i++){         //{2}
        for (var j=0; j<length-1; j++ ){ //{3}
            if (array[j] > array[j+1]){  //{4}
                swap(j, j+1);            //{5}
            }
        }
    }
};
```

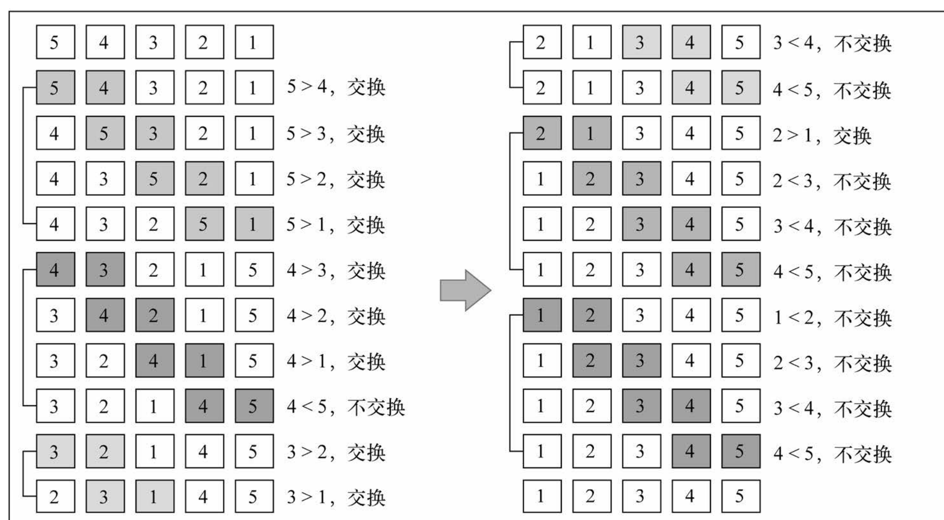
首先，声明一个名为length的变量，用来存储数组的长度（行{1}）。这一步可选，它能帮助我们在行{2}和行{3}时直接使用数组的长度。接着，外循环（行{2}）会从数组的第一位迭代至最后一位，它控制了在数组中经过多少轮排序（应该是数组中每项都经过一轮，轮数和数组长度一致）。然后，内循环将从第一位迭代至倒数第二位，内循环实际上进行当前项和下一项的比较（行{4}）。如果这两项顺序不对（当前项比下一项大），则交换它们（行{5}），意思是位置为j+1的值将会被换置到位置j处，反之亦然。

现在我们得声明swap函数（一个私有函数，只能用在ArrayList类的内部代码中）：

```
var swap = function(index1, index2){
    var aux = array[index1];
    array[index1] = array[index2];
    array[index2] = aux;
};
```

交换时，我们用一个中间值来存储某一交换项的值。其他排序法也会用到这个方法，因此我们声明一个方法放置这段交换代码以便重用。

下面这个示意图展示了冒泡排序的工作过程：



该示意图中每一小段表示外循环的一轮（行{2}），而相邻两项的比较则是在内循环中进行的（行{3}）。

我们将使用下面这段代码来测试冒泡排序算法，看结果是否和示意图所示一致：

```
function createNonSortedArray(size){ //{6}
    var array = new ArrayList();
    for (var i = size; i> 0; i--){
        array.insert(i);
    }
  return array;
}

var array = createNonSortedArray(5); //{7}
console.log(array.toString());       //{8}
array.bubbleSort();                  //{9}
console.log(array.toString());       //{10}
```

为了辅助测试本章将要学习的排序算法，我们将创建一个函数来自动地创建一个未排序的数组，数组的长度由函数参数指定（行{6}）。如果传递5作为参数，该函数会创建如下数组：[5, 4, 3, 2, 1]。调用这个函数并将返回值存储在一个变量中，该变量将包含这个以某些数字来初始化的ArrayList类实例（行{7}）。我们在控制台上输出这个数组内容，确保这是一个未排序数组（行{8}），接着我们调用冒泡排序方法（行{9}）并再次在控制台上输出数组内容以验证数组已被排序了（行{10}）。



你可以从书本的支持页面（或GitHub仓库）所下载的源码中找到完整的`ArrayList`源码和测试代码（带有补充注释的）。

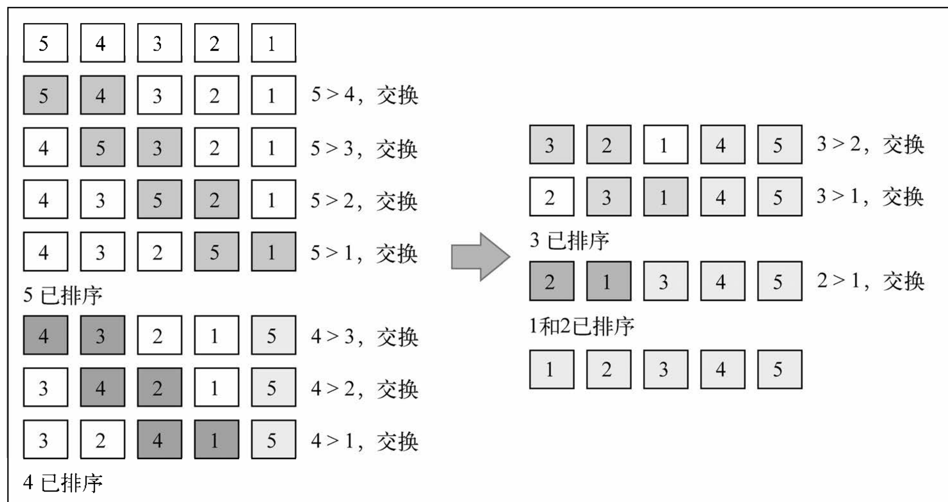
注意当算法执行外循环的第二轮的时候，数字4和5已经是正确排序的了。尽管如此，在后续比较中，它们还一直在进行着比较，即使这是不必要的。因此，我们可以稍稍改进一下冒泡排序算法。

改进后的冒泡排序

如果从内循环减去外循环中已跑过的轮数，就可以避免内循环中所有不必要的比较（行`{1}`）。

```
this.modifiedBubbleSort = function(){
    var length = array.length;
    for (var i=0; i<length; i++){
        for (var j=0; j<length-1-i; j++ ){ //{1}
            if (array[j] > array[j+1]){
                swap(j, j+1);
            }
        }
    }
};
```

下面这个示意图展示了改进后的冒泡排序算法是如何执行的：





注意已经在正确位置上的数字没有被比较。即便我们做了这个小改变，改进了一下冒泡排序算法，但我们还是不推荐该算法，它的复杂度是$O(n^2)$。

我们将在下一章对大O表示法做更多的讨论。

10.1.2　选择排序

选择排序算法是一种原址比较排序算法。选择排序大致的思路是找到数据结构中的最小值并将其放置在第一位，接着找到第二小的值并将其放在第二位，以此类推。

下面是选择排序算法的源代码：

```
this.selectionSort = function(){
    var length = array.length,            //{1}
        indexMin;
    for (var i=0; i<length-1; i++){       //{2}
        indexMin = i;                     //{3}
        for (var j=i; j<length; j++){     //{4}
            if(array[indexMin]>array[j]){ //{5}
                indexMin = j;             //{6}
            }
        }
        if (i !== indexMin){              //{7}
            swap(i, indexMin);
        }
    }
};
```

首先声明一些将在算法内使用的变量（行`{1}`）。接着，外循环（行`{2}`）迭代数组，并控制迭代轮次（数组的第n个值——下一个最小值）。我们假设本迭代轮次的第一个值为数组最小值（行`{3}`）。然后，从当前`i`的值开始至数组结束（行`{4}`），我们比较是否位置`j`的值比当前最小值小（行`{5}`）；如果是，则改变最小值至新最小值（行`{6}`）。当内循环结束（行`{4}`），将得出数组第n小的值。最后，如果该最小值和原最小值不同（行`{7}`），则交换其值。

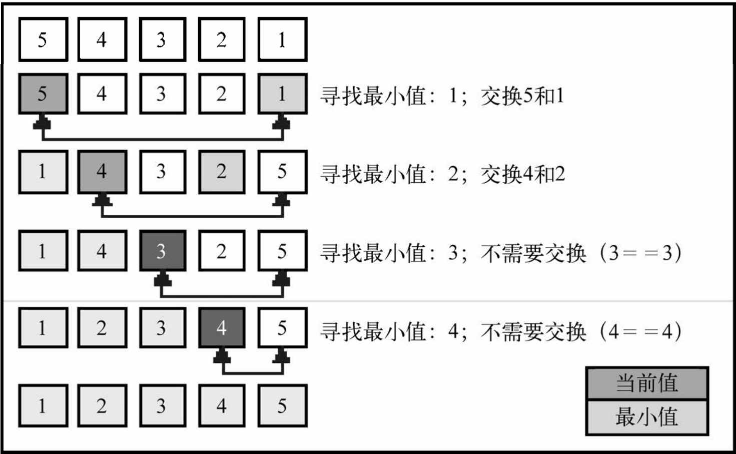
用以下代码段来测试选择排序算法：

```
array = createNonSortedArray(5);
console.log(array.toString());
array.selectionSort();
console.log(array.toString());
```

下面的示意图展示了选择排序算法，此例基于之前代码中所用的数组。

数组底部的箭头指示出当前迭代轮寻找最小值的数组范围（内循环`{4}`），示意图中的每一步则表示外循环。

选择排序同样也是一个复杂度为$O(n^2)$的算法。和冒泡排序一样，它包含有嵌套的两个循环，这导致了二次方的复杂度。然而，接下来要学的插入排序比选择排序性能要好。



10.1.3　插入排序

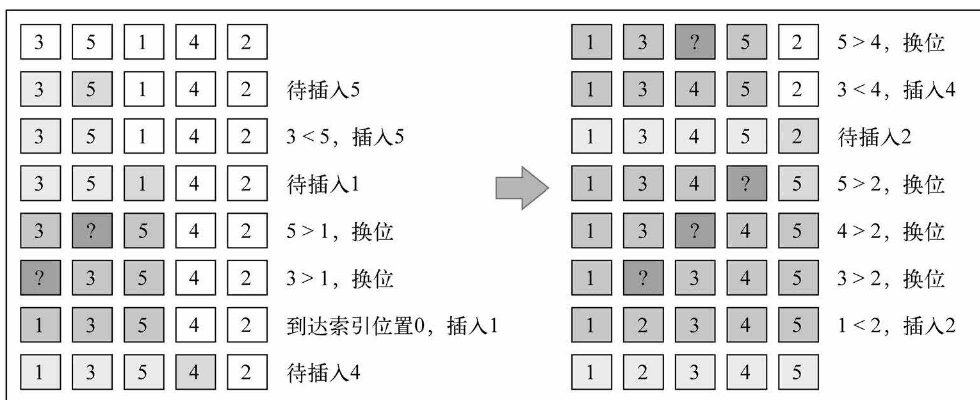
插入排序每次排一个数组项，以此方式构建最后的排序数组。假定第一项已经排序了，接着，它和第二项进行比较，第二项是应该待在原位还是插到第一项之前呢？这样，头两项就已正确排序，接着和第三项比较（它是该插入到第一、第二还是第三的位置呢？），以此类推。

下面这段代码表示插入排序算法：

```
this.insertionSort = function(){
    var length = array.length,            //{1}
        j, temp;
    for (var i=1; i<length; i++){         //{2}
        j = i;                            //{3}
        temp = array[i];                  //{4}
        while (j>0 && array[j-1] > temp){ //{5}
            array[j] = array[j-1];        //{6}
            j--;
        }
        array[j] = temp;                  //{7}
    }
};
```

照例，算法的第一行用来声明代码中使用的变量（行{1}）。接着，迭代数组来给第*i*项找到正确的位置（行{2}）。注意，算法是从第二个位置（索引1）而不是0位置开始的（我们认为第一项已排序了）。然后，用i的值来初始化一个辅助变量（行{3}）并将其值亦存储于一临时变量中（行{4}），便于之后将其插入到正确的位置上。下一步是要找到正确的位置来插入项目。只要变量j比0大（因为数组的第一个索引是0——没有负值的索引）并且数组中前面的值比待比较的值大（行{5}），我们就把这个值移到当前位置上（行{6}）并减小j。最终，该项目能插入到正确的位置上。

下面的示意图展示了一个插入排序的实例：



举个例子，假定待排序数组是[3, 5, 1, 4, 2]。这些值将被插入排序算法按照下面形容的步骤进行排序。

(1) 3已被排序，所以我们从数组第二个值5开始。3比5小，所以5待在原位（数组的第二位）。3和5排序完毕。

(2) 下一个待排序和插到正确位置上去的值是1（目前在数组的第三位）。5比1大，所以5被移至第三位去了。我们得分析1是否应该被插入到第二位——1比3大吗？不，所以3被移到第二位去了。接着，我们得证明1应该插入到数组的第一位上。因为0是第一个位置且没有负数位，所以1必须被插入到第一位。1、3、5三个数字已经排序。

(3) 4应该在当前位置（索引3）还是要移动到索引较低的位置上呢？4比5小，所以5移动到索引3位置上去。那么应该把4插到索引2的位置上去吗？4要比3大，所以4插入到数组的位置3上。

(4) 下一个待插入的数字是2（数组的位置4）。5比2大，所以5移动至索引4。4比2大，所以4也得移动（位置3）。3也比2大，所以3还得移动。1比2小，所以2插入到数组的第二位置上。至此，数组已排序完成。

排序小型数组时，此算法比选择排序和冒泡排序性能要好。

10.1.4　归并排序

归并排序是第一个可以被实际使用的排序算法。你在本书中学到的前三个排序算法性能不好，但归并排序性能不错，其复杂度为$O(n\log^n)$。



> JavaScript的Array类定义了一个sort函数（Array.prototype.sort）用以排序JavaScript数组（我们不必自己实现这个算法）。ECMAScript没有定义用哪个排序算法，所以浏览器厂商可以自行去实现算法。例如，Mozilla Firefox使用归并排序作为Array.prototype.sort的实现，而Chrome使用了一个快速排序（下面我们会学习的）的变体。

归并排序是一种分治算法。其思想是将原始数组切分成较小的数组，直到每个小数组只有一个位置，接着将小数组归并成较大的数组，直到最后只有一个排序完毕的大数组。

由于是分治法，归并排序也是递归的：

```
this.mergeSort = function(){
    array = mergeSortRec(array);
};
```

像之前的章节一样，每当要实现一个递归函数，我们都会实现一个实际被执行的辅助函数。对归并排序我们也会这么做。我们将声明mergeSort方法以供随后使用，而mergeSort方法将会调用mergeSortRec，该函数是一个递归函数：

```
var mergeSortRec = function(array){
    var length = array.length;
    if(length === 1) {      //{1}
        return array;       //{2}
    }
    var mid = Math.floor(length / 2),     //{3}
        left = array.slice(0, mid),       //{4}
        right = array.slice(mid, length); //{5}

    return merge(mergeSortRec(left), mergeSortRec(right)); //{6}
};
```

归并排序将一个大数组转化为多个小数组直到只有一个项。由于算法是递归的，我们需要一个停止条件，在这里此条件是判断数组的长度是否为1（行{1}）。如果是，则直接返回这个长度为1的数组（行{2}），因为它已排序了。

如果数组长度比1大，那么我们得将其分成小数组。为此，首先得找到数组的中间位（行{3}），找到后我们将数组分成两个小数组，分别叫作left（行{4}）和right（行{5}）。left数组由索引0至中间索引的元素组成，而right数组由中间索引至原始数组最后一个位置的元素组成。

下面的步骤是调用merge函数（行{6}），它负责合并和排序小数组来产生大数组，直到回到原始数组并已排序完成。为了不断将原始数组分成小数组，我们得再次对left数组和right数组递归调用mergeSortRec，并同时作为参数传递给merge函数。

```
var merge = function(left, right){
    var result = [], // {7}
        il = 0,
```

```
        ir = 0;

    while(il < left.length && ir < right.length) { // {8}
        if(left[il] < right[ir]) {
            result.push(left[il++]);  // {9}
        } else{
            result.push(right[ir++]); // {10}
        }
    }

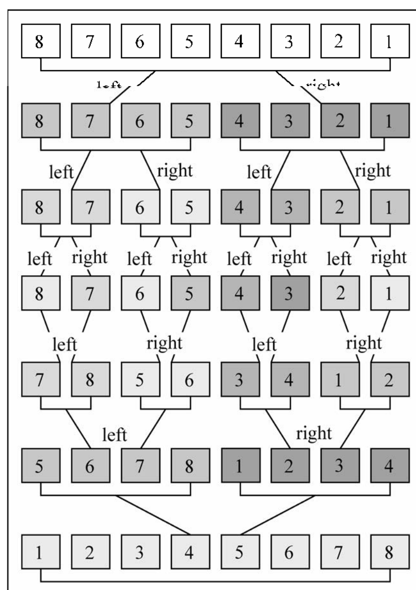
    while (il < left.length){     // {11}
         result.push(left[il++]);
    }

    while (ir < right.length){    // {12}
        result.push(right[ir++]);
    }

    return result; // {13}
};
```

merge函数接受两个数组作为参数，并将它们归并至一个大数组。排序发生在归并过程中。首先，需要声明归并过程要创建的新数组以及用来迭代两个数组（left和right数组）所需的两个变量（行{7}）。迭代两个数组的过程中（行{8}），我们比较来自left数组的项是否比来自right数组的项小。如果是，将该项从left数组添加至归并结果数组，并递增迭代数组的控制变量（行{9}）；否则，从right数组添加项并递增相应的迭代数组的控制变量（行{10}）。接下来，将left数组或者right数组所有剩余的项添加到归并数组中。最后，将归并数组作为结果返回。

如果执行mergeSort函数，下图是具体的执行过程：



可以看到，算法首先将原始数组分割直至只有一个元素的子数组，然后开始归并。归并过程也会完成排序，直至原始数组完全合并并完成排序。

10.1.5　快速排序

快速排序也许是最常用的排序算法了。它的复杂度为$O(n\log^n)$，且它的性能通常比其他的复杂度为$O(n\log^n)$的排序算法要好。和归并排序一样，快速排序也使用分治的方法，将原始数组分为较小的数组（但它没有像归并排序那样将它们分割开）。

快速排序比到目前为止你学过的其他排序算法要复杂一些。让我们一步步地来学习。

(1) 首先，从数组中选择中间一项作为主元。

(2) 创建两个指针，左边一个指向数组第一个项，右边一个指向数组最后一个项。移动左指针直到我们找到一个比主元大的元素，接着，移动右指针直到找到一个比主元小的元素，然后交换它们，重复这个过程，直到左指针超过了右指针。这个过程将使得比主元小的值都排在主元之前，而比主元大的值都排在主元之后。这一步叫作划分操作。

(3) 接着，算法对划分后的小数组（较主元小的值组成的子数组，以及较主元大的值组成的子数组）重复之前的两个步骤，直至数组已完全排序。

让我们开始快速排序的实现吧：

```
this.quickSort = function(){
    quick(array,  0, array.length - 1);
};
```

就像归并算法那样，开始我们声明一个主方法来调用递归函数，传递待排序数组，以及索引0及其最末的位置（因为我们要排整个数组，而不是一个子数组）作为参数。

```
var quick = function(array, left, right){

    var index; //{1}

    if (array.length > 1) { //{2}

        index = partition(array, left, right); //{3}

        if (left < index - 1) {                //{4}
            quick(array, left, index - 1);     //{5}
        }

        if (index < right) {  //{6}
            quick(array, index, right);        //{7}
        }
    }
};
```

首先声明index（行{1}），该变量能帮助我们将子数组分离为较小值数组和较大值数组，这样，我们就能再次递归的调用quick函数了。partition函数返回值将赋值给index（行{3}）。

如果数组的长度比1大（因为只有一个元素的数组必然是已排序了的（行{2}），我们将对给定子数组执行partition操作（第一次调用是针对整个数组）以得到index（行{3}）。如果子数组存在较小值的元素（行{4}），则对该数组重复这个过程（行{5}）。同理，对存在较大值得子数组也是如此，如果存在子数组存在较大值，我们也将重复快速排序过程（行{7}）。

1. 划分过程

现在，让我们看看划分过程：

```
var partition = function(array, left, right) {

    var pivot = array[Math.floor((right + left) / 2)], //{8}
        i = left,                                      //{9}
        j = right;                                     //{10}

    while (i <= j) {                //{11}
        while (array[i] < pivot) {  //{12}
            i++;
        }
        while (array[j] > pivot) {  //{13}
            j--;
        }
        if (i <= j) { //{14}
            swapQuickStort(array, i, j); //{15}
            i++;
            j--;
        }
    }
    return i; //{16}
};
```

第一件要做的事情是选择主元（pivot），有好几种方式。最简单的一种是选择数组的第一项（最左项）。然而，研究表明对于几乎已排序的数组，这不是一个好的选择，它将导致该算法的最差表现。另外一种方式是随机选择一个数组项或是选择中间项。在本实现中，我们选择中间项作为主元（行{8}）。我们初始化两个指针：left（低——行{9}），初始化为数组第一个元素；right（高——行{10}），初始化为数组最后一个元素。

只要left和right指针没有相互交错（行{11}），就执行划分操作。首先，移动left指针直到找到一个元素比主元大（行{12}）。对right指针，我们做同样的事情，移动right指针直到我们找到一个元素比主元小。

当左指针指向的元素比主元大且右指针指向的元素比主元小，并且此时左指针索引没有右指针索引大（行{14}），意思是左项比右项大（值比较）。我们交换它们，然后移动两个指针，并重复此过程（从行{11}再次开始）。

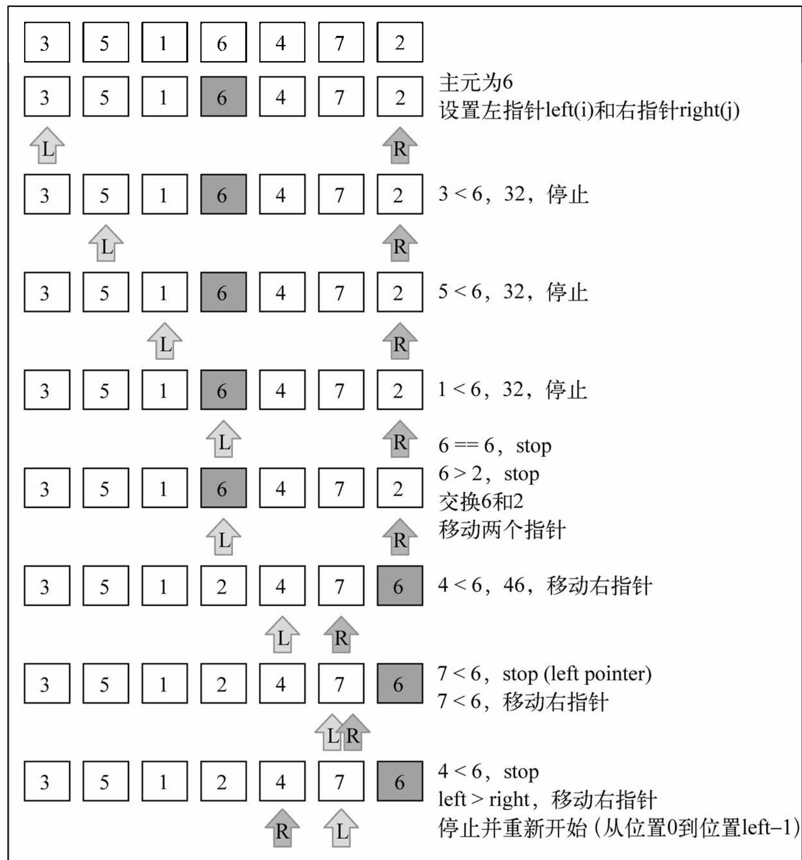
在划分操作结束后，返回左指针的索引，用来在行{3}处创建子数组。

swapQuickStort函数和我们在本章开始处实现的swap函数十分相似。唯一的不同之处是发生交换值的的数组同样也是函数的参数。

```
var swapQuickStort = function(array, index1, index2){
    var aux = array[index1];
    array[index1] = array[index2];
    array[index2] = aux;
};
```

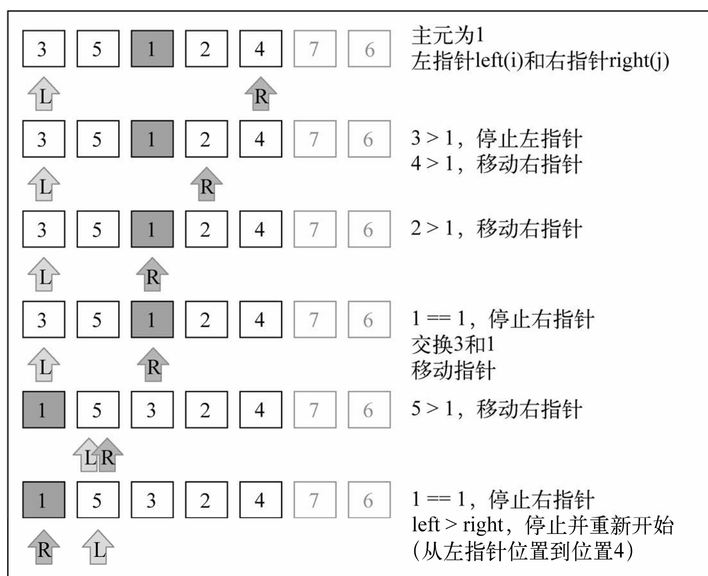
2. 快速排序实战

让我来一步步地看一个快速排序的实际例子：

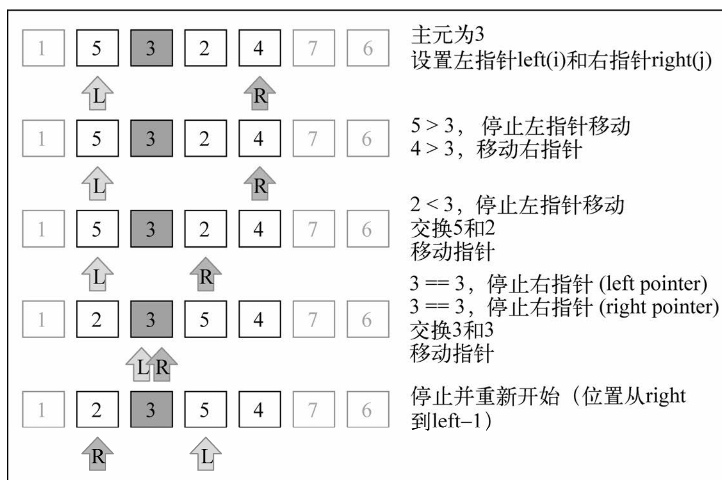


给定数组[3, 5, 1, 6, 4, 7, 2]，前面的示意图展示了划分操作的第一次执行。

下面的示意图展示了对有较小值的子数组执行的划分操作（注意7和6不包含在子数组之内）：



接着，我们继续创建子数组，请看下图，但是这次操作是针对上图中有较大值的子数组（有1的那个较小子数组不用再划分了，因为它仅含有一个项）。



子数组（`[2, 3, 5, 4]`）中的较小子数组（`[2, 3]`）继续进行划分（算法代码中的行`{5}`）：



然后子数组（[2， 3， 5， 4]）中的较大子数组（[5， 4]）也继续进行划分（算法中的行{7}），示意图如下：



最终，较大子数组[6，7]也会进行划分（partition）操作，快速排序算法的操作执行完成。

10.2　搜索算法

现在，让我们来谈谈搜索算法。回顾一下之前章节所实现的算法，我们会发现BinarySearchTree类的search方法（第8章），以及LinkedList类的indexOf方法（第5章）等，都是搜索算法，当然，它们每一个都是根据其各自的数据结构来实现的。所以，我们已经熟悉两个搜索算法了，只是还不知道它们“正式”的名称而已。

10.2.1 顺序搜索

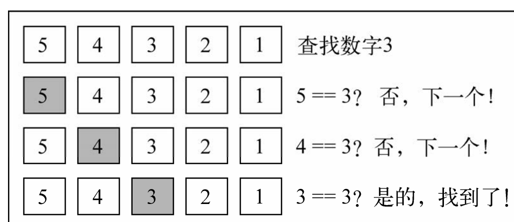
顺序或线性搜索是最基本的搜索算法。它的机制是，将每一个数据结构中的元素和我们要找的元素做比较。顺序搜索是最低效的一种搜索算法。

以下是其实现：

```
this.sequentialSearch = function(item){
    for (var i=0; i<array.length; i++){ //{1}
        if (item === array[i])          //{2}
            return i;                   //{3}
        }
    }
    return -1;  //{4}
};
```

顺序搜索迭代整个数组（行{1}），并将每个数组元素和搜索项作比较（行{2}）。如果搜索到了，算法将用返回值来标示搜索成功。返回值可以是该搜索项本身，或是true，又或是搜索项的索引（行{3}）。如果没有找到该项，则返回-1（行{4}），表示该索引不存在；也可以考虑返回false或者null。

假定有数组[5, 4, 3, 2, 1]和待搜索值3，下图展示了顺序搜索的示意图：



10.2.2 二分搜索

二分搜索算法的原理和猜数字游戏类似，就是那个有人说“我正想着一个1到100的数字”的游戏。我们每回应一个数字，那个人就会说这个数字是高了、低了还是对了。

这个算法要求被搜索的数据结构已排序。以下是该算法遵循的步骤。

(1) 选择数组的中间值。
(2) 如果选中值是待搜索值，那么算法执行完毕（值找到了）。
(3) 如果待搜索值比选中值要小，则返回步骤1并在选中值左边的子数组中寻找。
(4) 如果待搜索值比选中值要大，则返回步骤1并在选种值右边的子数组中寻找。

以下是其实现：

```
this.binarySearch = function(item){
    this.quickSort(); //{1}

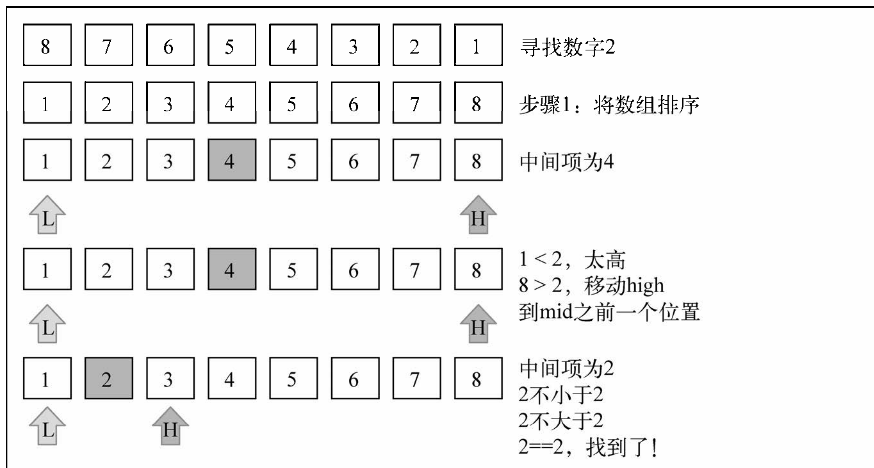
    var low = 0,                 //{2}
        high = array.length - 1, //{3}
        mid, element;

    while (low <= high){ //{4}
        mid = Math.floor((low + high) / 2); //{5}
        element = array[mid];               //{6}
        if (element < item) {               //{7}
            low = mid + 1;                  //{8}
        } else if (element > item) {        //{9}
            high = mid - 1;                 //{10}
        } else {
            return mid;                     //{11}
        }
    }
    return -1; //{12}
};
```

开始前需要先将数组排序，我们可以选择任何一个在10.1节中实现的排序算法。这里我们选择了快速排序。在数组排序之后，我们设置`low`（行`{2}`）和`high`（行`{3}`）指针（它们是边界）。

当`low`比`high`小时（行`{4}`），我们计算得到中间项索引并取得中间项的值，此处如果`low`比`high`大，则意思是该待搜索值不存在并返回`-1`（行`{12}`）。接着，我们比较选中项的值和搜索值（行`{7}`）。如果小了，则选择数组低半边并重新开始。如果选中项的值比搜索值大了，则选择数组高半边并重新开始。若两者都是不是，则意味着选中项的值和搜索值相等，因此，直接返回该索引（行`{11}`）。

给定下图所示数组，让我们试试看搜索2。这些是算法将会执行的步骤：



第8章中，我们实现的`BinarySearchTree`类有一个`search`方法，和这个二分搜索完全一样，只不过它是针对树数据结构的。

10.3 小结

本章介绍了搜索和排序算法。你学到了冒泡、选择、插入、归并以及快速排序算法，它们是用来排序数据结构的。你还学到了顺序搜索和二分搜索（需要数据结构已排序）。

本章中你所掌握的方法可以运用到任何数据结构或数据类型上，你只需在源代码上做一些必要的修改即可。

下一章，你将学习算法中一些更高级的技术，以及本章中提及的大O表示法。

第 11 章

算法补充知识

到现在为止，我们愉快地学习了不同的数据结构的实现，其中包括常用的排序和搜索算法。算法和编程的世界很有意思。本章，你会进一步了解这个世界，并且我们将探讨进一步深入其中的途径（如果你感兴趣的话）。

我们将介绍第8章中提到过的*递归*。还将学习*动态规划*、*贪心算法*，以及众所周知的*大O表示法*（第10章提到过）。此外，我们还会介绍如何用算法娱乐身心，提升我们的知识水平，以达到增强编程和解决问题的能力的目的。

11.1 递归

*递归*是一种解决问题的方法，它解决问题的各个小部分，直到解决最初的大问题。通常涉及函数调用自身。

能够像下面这样直接调用自身的方法或函数，是递归函数：

```
var recursiveFunction = function(someParam){
  recursiveFunction(someParam);
};
```

能够像下面这样间接调用自身的函数，也是递归函数：

```
var recursiveFunction1 = function(someParam){
  recursiveFunction2(someParam);
};
var recursiveFunction2 = function(someParam){
  recursiveFunction1(someParam);
};
```

假设现在必须要执行`recursiveFunction`，结果是什么？单单就上述情况而言，它会一直执行下去。因此，每个递归函数都必须要有边界条件，即一个不再递归调用的条件（停止点），以防止无限递归。

11.1.1 JavaScript 调用栈大小的限制

如果忘记加上用以停止函数递归调用的边界条件，会发生什么呢？递归并不会无限地执行下去；浏览器会抛出错误，也就是所谓的栈溢出错误（stack overflow error）。

每个浏览器都有自己的上限，可用以下代码测试：

```
var i = 0;

function recursiveFn () {
    i++;
    recursiveFn();
}

try {
    recursiveFn();
} catch (ex) {
    alert('i = ' + i + ' error: ' + ex);
}
```

在Chrome v37中，这个函数执行了20 955次，而后浏览器抛出错误`RangeError: Maximum call stack size exceeded`（超限错误：超过最大调用栈大小）。Firefox v27中，函数执行了343 429次，然后浏览器抛出错误 `InternalError: too much recursion`（内部错误：递归次数过多）。

> 根据操作系统和浏览器的不同，具体数值会所有不同，但区别不大。

ECMAScript 6有尾调用优化（tail call optimization）。如果函数内最后一个操作是调用函数（就像示例中加粗的那行），会通过“跳转指令”（jump） 而不是“子程序调用”（subroutine call）来控制。也就是说，在ECMAScript 6中，这里的代码可以一直执行下去。所以，具有停止递归的边界条件非常重要。

> 有关尾调用优化的更多相关信息，请访问http://goo.gl/ZdTZzg。

11.1.2 斐波那契数列

回到第10章提到的斐波那契问题。斐波那契数列的定义如下：

- 1和2的斐波那契数是 1；
- n（n>2）的斐波那契数是(n–1)的斐波那契数加上(n–2)的斐波那契数。

那么，让我们开始实现斐波那契函数：

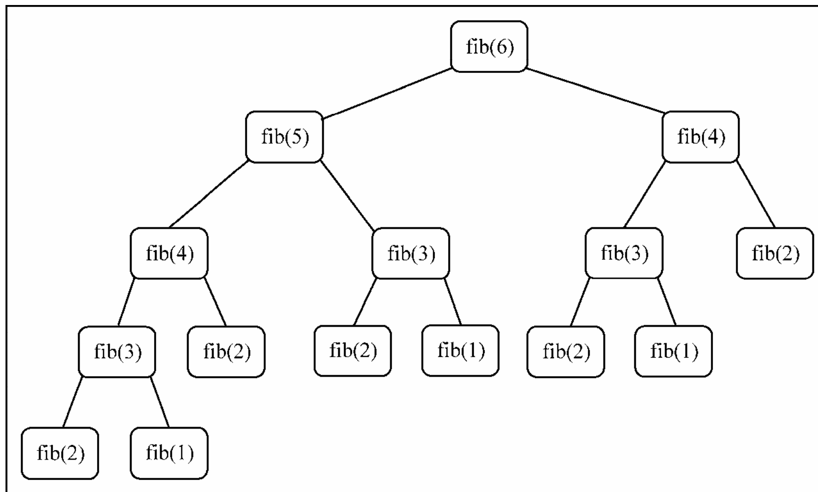
```
function fibonacci(num){
    if (num === 1 || num === 2){ //{1}
        return 1;
    }
}
```

边界条件是已知的，1和2的斐波那契数（行`{1}`）是1。现在，让我们完成斐波那契函数的实现：

```
function fibonacci(num){
    if (num === 1 || num === 2){
        return 1;
    }
    return fibonacci(num - 1) + fibonacci(num - 2);
}
```

我们已经知道，当n大于2时，Fibonacci(n)等于Fibonacci(n–1)+Fibonacci(n–2)。

现在，斐波那契函数实现完毕。让我们试着找出6的斐波那契数，其会产生如下函数调用：



我们也可以用非递归的方式实现斐波那契函数：

```
function fib(num){
    var n1 = 1,
        n2 = 1,
        n = 1;
    for (var i = 3; i<=num; i++){
        n = n1 + n2;
        n1 = n2;
        n2 = n;
    }
    return n;
}
```

为何用递归呢？更快吗？递归并不比普通版本更快，反倒更慢。但要知道，递归更容易理解，并且它所需的代码量更少。



ECMAScript 6中，因为尾调用优化的缘故，递归并不会更慢。但是在其他语言中，递归通常更慢。

所以，我们用递归，通常是因为它更容易解决问题。

11.2 动态规划

动态规划（Dynamic Programming，DP）是一种将复杂问题分解成更小的子问题来解决的优化技术。

本书之前提到过几次动态规划技术。用动态规划解决的一个问题是第9章中的深度优先搜索。



要注意动态规划和分而治之（归并排序和快速排序算法中用到的那种）是不同的方法。分而治之方法是把问题分解成相互独立的子问题，然后组合它们的答案，而动态规划则是将问题分解成相互依赖的子问题。

另一个例子是上一节解决的斐波那契问题。我们将斐波那契问题分解成如该节图示的小问题。

用动态规划解决问题时，要遵循三个重要步骤：

(1) 定义子问题；
(2) 实现要反复执行而解决子问题的部分（这一步要参考前一节讨论的递归的步骤）；
(3) 识别并求解出边界条件。

能用动态规划解决的一些著名的问题如下。

- 背包问题：给出一组项目，各自有值和容量，目标是找出总值最大的项目的集合。这个问题的限制是，总容量必须小于等于“背包”的容量。
- 最长公共子序列：找出一组序列的最长公共子序列（可由另一序列删除元素但不改变余下元素的顺序而得到）。
- 矩阵链相乘：给出一系列矩阵，目标是找到这些矩阵相乘的最高效办法（计算次数尽可能少）。相乘操作不会进行，解决方案是找到这些矩阵各自相乘的顺序。
- 硬币找零：给出面额为$d_1 \cdots d_n$的一定数量的硬币和要找零的钱数，找出有多少种找零的方法。
- 图的全源最短路径：对所有顶点对(u, v)，找出从顶点u到顶点v的最短路径。

接下来的例子，涉及硬币找零问题的一个变种。

最少硬币找零问题

最少硬币找零问题是硬币找零问题的一个变种。硬币找零问题是给出要找零的钱数，以及可用的硬币面额$d_1 \cdots d_n$及其数量，找出有多少种找零方法。最少硬币找零问题是给出要找零的钱数，以及可用的硬币面额$d_1 \cdots d_n$及其数量，找到所需的最少的硬币个数。

例如，美国有以下面额（硬币）：$d_1=1$，$d_2=5$，$d_3=10$，$d_4=25$。

如果要找36美分的零钱，我们可以用1个25美分、1个10美分和1个便士（1美分）。

如何将这个解答转化成算法？

最少硬币找零的解决方案是找到n所需的最小硬币数。但要做到这一点，首先得找到对每个$x<n$的解。然后，我们将解建立在更小的值的解的基础上。

来看看算法：

```
function MinCoinChange(coins){
    var coins = coins; //{1}
    var cache = {};    //{2}

    this.makeChange = function(amount) {
        var me = this;
        if (!amount) { //{3}
            return [];
        }
        if (cache[amount]) { //{4}
            return cache[amount];
        }
        var min = [], newMin, newAmount;
        for (var i=0; i<coins.length; i++){ //{5}
            var coin = coins[i];
            newAmount = amount - coin; //{6}
            if (newAmount >= 0){
                newMin = me.makeChange(newAmount); //{7}
            }
            if (
                newAmount >= 0 && //{8}
                (newMin.length < min.length-1 || !min.length)//{9}
                && (newMin.length || !newAmount) //{10}
                ){
                min = [coin].concat(newMin); //{11}
                console.log('new Min ' + min + ' for ' + amount);
            }
        }
        return (cache[amount] = min); //{12}
    };
}
```

为了更有条理，我们创建了一个类，解决给定面额的最少硬币找零问题。让我们一步步解读

这个算法。

MinCoinChange类接收coins参数（行{1}），代表问题中的面额。对美国的硬币系统而言，它是[1, 5, 10, 25]。我们可以随心所欲传递任何面额。此外，为了更加高效且不重复计算值，我们使用了cache（行{2}）。

接下来是makeChange方法，它也是一个递归函数，找零问题由它解决。首先，若amount不为正（< 0），就返回空数组（行{3}）；方法执行结束后，会返回一个数组，包含用来找零的各个面额的硬币数量（最少硬币数）。接着，检查cache缓存。若结果已缓存（行{4}），则直接返回结果；否则，执行算法。

我们基于coins参数（面额）解决问题。因此，对每个面额（行{5}），我们都计算newAmount（行{6}）的值，它的值会一直减小，直到能找零的最小钱数（别忘了本算法对所有的x < amount都会计算makeChange结果）。若newAmount是合理的值（正值），我们也会计算它的找零结果（行{7}）。

最后，我们判断newAmount是否有效，minValue（最少硬币数）是否是最优解，与此同时minValue和newAmount是否是合理的值（{行10}）。若以上判断都成立，意味着有一个比之前更优的答案（行{11}，以5美分为例，可以给5便士或者1个5美分镍币，1个5美分镍币是最优解）。最后，返回最终结果（行{12}）。

测试一下这个算法：

```
var minCoinChange = new MinCoinChange([1, 5, 10, 25]);
console.log(minCoinChange.makeChange(36));
```

要知道，如果我们检查cache变量，会发现它存储了从1到36美分的所有结果。以上代码的结果是[1, 10, 25]。

本书的源码中会有几行多余代码，输出算法的步骤。例如，使用面额[1, 3, 4]，并对钱数6执行算法，会产生以下输出：

```
new Min 1 for 1
new Min 1,1 for 2
new Min 1,1,1 for 3
new Min 3 for 3
new Min 1,3 for 4
new Min 4 for 4
new Min 1,4 for 5
new Min 1,1,4 for 6
new Min 3,3 for 6
[3, 3]
```

所以，找零钱数为6时，最佳答案是两枚价值为3的硬币。

11.3　贪心算法

贪心算法遵循一种近似解决问题的技术，期盼通过每个阶段的局部最优选择（当前最好的解），从而达到全局的最优（全局最优解）。它不像动态规划那样计算更大的格局。

我们使用上一节中的同一个题目，以便看到区别。

最少硬币找零问题

最少硬币找零问题也能用贪心算法解决。大部分情况的结果是最优的，不过对有些面额而言，结果不会是最优的。

来看看算法：

```
function MinCoinChange(coins){
    var coins = coins; //{1}

    this.makeChange = function(amount) {
        var change = [],
            total = 0;
        for (var i=coins.length; i>=0; i--){ //{2}
            var coin = coins[i];
            while (total + coin <= amount) { //{3}
                change.push(coin);           //{4}
                total += coin;               //{5}
            }
        }
        return change;
    };
}
```

不得不说贪心版本的`MinCoinChange`比DP版本的简单多了。和动态规划方法相似，我们传递面额参数，实例化`MinCoinChange`（行`{1}`）。

对每个面额（行`{2}`——从大到小），把它的值和`total`相加后，`total`需要小于`amount`（行`{3}`）。我们会将当前面额`coin`添加到结果中（行`{4}`），也会将它和`total`相加（行`{5}`）。

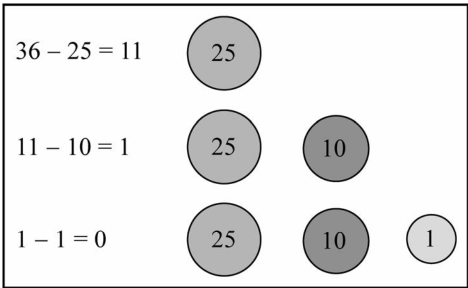
如你所见，这个解法很简单。从最大面额的硬币开始，拿尽可能多的这种硬币找零。当无法再拿更多这种价值的硬币时，开始拿第二大价值的硬币，依次继续。

用和DP方法同样的测试代码测试：

```
var minCoinChange = new MinCoinChange([1, 5, 10, 25]);
console.log(minCoinChange.makeChange(36));
```

结果依然是`[25, 10, 1]`，和用DP得到的一样。下图阐释了算法的执行过程：

然而，如果用`[1, 3, 4]`面额执行贪心算法，会得到结果`[4, 1, 1]`。如果用动态规划的解法，会得到最优的结果`[3, 3]`。



比起动态规划算法而言，贪心算法更简单、更快。然而，如我们所见，它并不总是得到最优答案。但是综合来看，它相对执行时间来说，输出了一个可以接受的解。

11.4　大 O 表示法

第10章引入了大O表示法的概念。它的确切含义是什么？是描述算法的性能和复杂程度。

分析算法时，时常遇到以下几类函数：

符　号	名　称
$O(1)$	常数的
$O(\log(n))$	对数的
$O((\log(n))c)$	对数多项式的
$O(n)$	线性的
$O(n^2)$	二次的
$O(n^c)$	多项式的
$O(c^n)$	指数的

11.4.1　理解大 O 表示法

如何衡量算法的效率？通常是用资源，例如CPU（时间）占用、内存占用、硬盘占用和网络占用。当讨论大O表示法时，一般考虑的是CPU（时间）占用。

让我们试着用一些例子来理解大O表示法的规则。

1. $O(1)$

考虑以下函数：

```
function increment(num){
    return ++num;
}
```

假设运行`increment(1)`函数，执行时间等于X。如果再用不同的参数（例如2）运行一次

increment函数，执行时间依然是X。和参数无关，increment函数的性能都一样。因此，我们说上述函数的复杂度是$O(1)$（常数）。

2. $O(n)$

现在以第10章中实现的顺序搜索算法为例：

```
function sequentialSearch(array, item){
    for (var i=0; i<array.length; i++){
        if (item === array[i]){ //{1}
            return i;
        }
    }
    return -1;
}
```

如果将含10个元素的数组（[1, ..., 10]）传递给该函数，假如搜索1这个元素，那么，第一次判断时就能找到想要搜索的元素。在这里我们假设每执行一次行{1}，开销是 1。

现在，假如要搜索元素11。行{1}会执行10次（遍历数组中所有的值，并且找不到要搜索的元素，因而结果返回 -1）。如果行{1}的开销是1，那么它执行10次的开销就是10，10倍于第一种假设。

现在，假如该数组有1000个元素（[1, ..., 1000]）。搜索1001的结果是行{1}执行了1000次（然后返回-1）。

注意，sequentialSearch函数执行的总开销取决于数组元素的个数（数组大小），而且也和搜索的值有关。如果是查找数组中存在的值，行{1}会执行几次呢？如果查找的是数组中不存在的值，那么行{1}就会执行和数组大小一样多次，这就是通常所说的最坏情况。

最坏情况下，如果数组大小是10，开销就是10；如果数组大小是1000，开销就是1000。可以得出sequentialSearch函数的时间复杂度是$O(n)$，n是（输入）数组的大小。

回到之前的例子，修改一下算法的实现，使之计算开销：

```
function sequentialSearch(array, item){
    var cost = 0;
    for (var i=0; i<array.length; i++){
        cost++;
        if (item === array[i]){ //{1}
            return i;
        }
    }
    console.log('cost for sequentialSearch with input size ' + array.length + ' is '
+ cost);
    return -1;
}
```

用不同大小的输入数组执行以上算法，可以看到不同的输出。

3. $O(n^2)$

用冒泡排序做$O(n^2)$的例子：

```
function swap(array, index1, index2){
    var aux = array[index1];
    array[index1] = array[index2];
    array[index2] = aux;
}
function bubbleSort(array){
    var length = array.length;
    for (var i=0; i<length; i++){        //{1}
        for (var j=0; j<length-1; j++ ){ //{2}
            if (array[j] > array[j+1]){
                swap(array, j, j+1);
            }
        }
    }
}
```

假设行`{1}`和行`{2}`的开销分别是1。修改算法的实现使之计算开销：

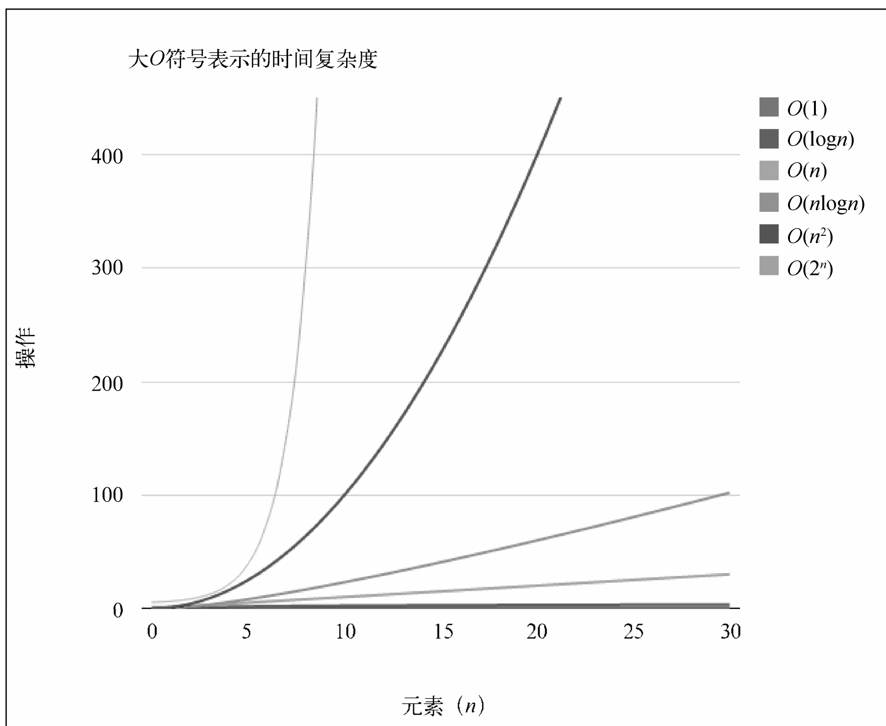
```
function bubbleSort(array){
    var length = array.length;
    var cost = 0;
    for (var i=0; i<length; i++){ //{1}
        cost++;
        for (var j=0; j<length-1; j++ ){ //{2}
            cost++;
            if (array[j] > array[j+1]){
                swap(array, j, j+1);
            }
        }
    }
    console.log('cost for bubbleSort with input size ' + length + ' is
' + cost);
}
```

如果用大小为10的数组执行`bubbleSort`，开销是 100（10^2）。如果用大小为100的数组执行`bubbleSort`，开销就是 10 000（100^2）。需要注意，我们每次增加输入的大小，执行都会越来越久。

> 时间复杂度$O(n)$的代码只有一层循环，而$O(n^2)$的代码有双层嵌套循环。如果算法有三层遍历数组的嵌套循环，它的时间复杂度很可能就是$O(n^3)$。

11.4.2 时间复杂度比较

下图比较了前述各个大O符号表示的时间复杂度：



这个图表是用JavaScript绘制的哦！在本书示例代码中，你可以到Chapter11下的bigOChart目录中找到绘制本图表的源代码。

在附录中，你可以找到本书实现的所有算法的时间复杂度的速查表。

11.5　用算法娱乐身心

我们学习算法并不单单是因为它是大学必修课，也不单单是因为我们想成为开发者。通过用在本书中学到的算法来解决问题，我们可以提高解决问题的能力，进而成为更棒的专业人士。

增长（解题）知识的最好方式是练习，而练习不一定是枯燥的。本节将展示一些网站，你可以访问它们并尝试从算法中获到快乐（甚至小赚一笔）。

这里列出一些有用的网站（有些不支持用JavaScript提交解答，但是我们依然可以将从本书中所学到的逻辑应用到其他语言上）。

- **UVa Online Judge**（http://uva.onlinejudge.org/）：这个网站包含了世界各大赛事的题目，包括由IBM赞助的ACM国际大学生程序竞赛（ICPC。若你依然在校，应尽量参与这项赛

事，如果团队获胜，则有可能免费享受一次国际旅行）。这个网站包括了成百上千的题目，可以应用本书所学的算法。

- **Sphere Online Judge**（http://www.spoj.com/）：这个网站和UVa Online Judge差不多，但支持用更多语言解题（包括JavaScript）。
- **Coder Byte**（http://coderbyte.com/）：这个网站包含了74个可以用JavaScript解答的题目（简单、中等难度和非常困难）。
- **Project Euler**（https://projecteuler.net/）：这个网站包含了一系列数学/计算机的编程题目。你所要做的就是输入那些题目的答案，不过我们可以用算法来找到正确的解答。
- **Hacker Rank**（https://www.hackerrank.com）：这个网站包含了263个挑战，分为16个类别（可以应用本书中的算法和更多其他算法）。它也支持JavaScript和其他语言。
- **Code Chef**（http://www.codechef.com/）：这个网站包含一些题目，并会举办在线比赛。
- **Top Coder**（http://www.topcoder.com/）：此网站会举办算法联赛，这些联赛通常由NASA、Google、Yahoo!、Amazon和Facebook这样的公司赞助。参加其中一些赛事，你可以获得到赞助公司工作的机会，而参与另一些赛事会赢得奖金。这个网站也提供很棒的解题和算法教程。

以上网站的另一个好处是，它们通常给出的是真实世界中的问题，而我们需要鉴别用哪一个算法解决它。通过这样的方式也能让我们明白本书中的算法并非局限于学术，而是能应用到现实问题上。

如果你想从事技术工作，强烈推荐你创建一个免费的GitHub（https://github.com）账号，你可以将上述网站的解答代码提交上去。如果你没有任何专业经验，GitHub可以帮助你建立一个作品集，还会对你找到第一份工作有帮助！

11.6 小结

本章进一步学习了递归，以及它在用动态规划解题时的应用。此外，我们还学习了贪心算法的相关知识。

我们介绍了大*O*表示法，以及如何运用它计算算法的时间复杂度。

另外我们还列出了一些网站。你可以免费注册这些网站，并应用你从本书中学到的知识，甚至还可能得到第一份IT行业的工作！

编程快乐！

附录 A

时间复杂度速查表



本书中实现了一些算法，本附录罗列了它们的时间复杂度，用大O符号表示。

A.1 数据结构

本书的主题是一些常用的数据结构。下表是它们插入、删除和搜索操作的时间复杂度：

数据结构	一般情况			最差情况		
	插入	删除	搜索	插入	删除	搜索
数组/栈/队列	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
链表	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$\mathrm{O}(n)$
双向链表	$O(1)$	$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
散列表	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
二分搜索树	$O(\log(n))$	$O(\log(n))$	$O(\log(n))$	$O(n)$	$O(n)$	$O(n)$

A.2 图

第9章提到两种表示图的方式。下表分别列出了使用这两种方式时，图的存储空间大小，及其增加顶点、增加边、删除顶点、删除边、查找顶点的时间复杂度：

节点/边的管理方式	存储空间	增加顶点	增加边	删除顶点	删除边	轮　询
相邻列表	$O(\lvert V\rvert+\lvert E\rvert)$	$O(1)$	$O(1)$	$O(\lvert V\rvert+\lvert E\rvert)$	$O(\lvert E\rvert)$	$O(\lvert V\rvert)$
相邻矩阵	$O(\lvert V\rvert^2)$	$O(\lvert V\rvert^2)$	$O(1)$	$O(\lvert V\rvert^2)$	$O(1)$	$O(1)$

A.3 排序算法

第10章介绍了一些常用的排序算法，以下是它们在最好、一般和最差的情况下的时间复杂度：

算法（用于数组）	时间复杂度		
	最好情况	一般情况	最差情况
冒泡排序	$O(n)$	$O(n^2)$	$O(n^2)$
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$
插入排序	$O(n)$	$O(n^2)$	$O(n^2)$
归并排序	$O(n\log(n))$	$O(n\log(n))$	$O(n\log(n))$
快速排序	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$

A.4　搜索算法

下表整理了本书中的搜索算法的时间复杂度，包括图的遍历算法：

算　　法	数据结构	最差情况
顺序搜索	数组和链表	$O(n)$
二分搜索	排好序的数组或二分搜索树	$O(\log(n))$
深度优先搜索（DPS）	|V|为顶点而|E|为边的图	$O(\|V\|+\|E\|)$
广度优先搜索（BFS）	|V|为顶点而|E|为边的图	$O(\|V\|+\|E\|)$

致　　谢

我要感谢父母这些年来对我的教育、指导和建议，帮助我成为一个更好、更专业的人。特别要感谢我的丈夫给予我的耐心、支持和鼓励。

我也要感谢我在FAESA的教授，是他们教会了我这本书中介绍的算法和数据结构。

我还要感谢朋友和读者的支持。我们通过会议相识，他们在社交网络上说看过我写的书，并给我反馈，这真的很好。非常感谢！

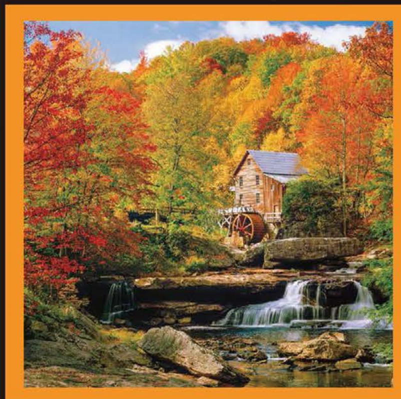
数据结构是计算机为了高效地利用资源而组织数据的一种方式。数据结构和算法是解决一切编程问题的基础。

本书首先介绍了JavaScript语言的基础知识，接着讨论了数组、队列、栈和链表等最重要的数据结构，接下来深入分析了散列表、字典和集合的工作原理，然后阐述了什么是树以及如何使用二叉树和二叉搜索树。之后，你还会学到图、DFS和BFS算法，学会如何区分顺序搜索、二分搜索、快速排序、冒泡排序等各种搜索和排序算法，以及如何实现它们。本书最后还介绍了动态规划和贪心算法等高级算法。

如果你是一名JavaScript开发者或者具备JavaScript的基础知识，并且想探索它的最佳能力，这本快节奏的书绝对适合你。要开始享受算法的乐趣，你只需要了解编程逻辑。

你将从本书中学到：

- 在数组、栈和队列中声明、初始化、添加和删除元素；
- 创建和使用最复杂的数据结构——图，以及DFS和BFS算法；
- 链表、双向链表和循环链表的作用；
- 用散列表、字典和集合存储不重复的元素；
- 二叉树和二叉搜索树的应用；
- 使用冒泡排序、选择排序、插入排序、归并排序和快速排序算法，对数据结构排序；
- 使用顺序搜索和二分搜索，搜索数据结构中的元素；
- 理解大*O*表示法、动态规划和贪婪算法的重要性。



[PACKT] PUBLISHING

图灵社区：iTuring.cn
热线：(010)51095186转600
分类建议 计算机/ Web开发/JavaScript
人民邮电出版社网址：www.ptpress.com.cn



ISBN 978-7-115-40414-5
定价：39.00元

看完了

如果您对本书内容有疑问，可发邮件至contact@turingbook.com，会有编辑或作译者协助答疑。也可访问图灵社区，参与本书讨论。

如果是有关电子书的建议或问题，请联系专用客服邮箱：ebook@turingbook.com。

在这里可以找到我们：

微博 @图灵教育 : 好书、活动每日播报
微博 @图灵社区 : 电子书和好文章的消息
微博 @图灵新知 : 图灵教育的科普小组
微信 图灵访谈 : ituring_interview，讲述码农精彩人生
微信 图灵教育 : turingbooks